Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Symbolische Ausführung für die Analyse von szenariobasierten Spezifikationen

Masterarbeit

im Studiengang Informatik

von

Timo Gutjahr

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 30.09.2016

Zusammenfassung

ScenarioTools ist ein Programm zur Erstellung und Analyse von szenariobasierten Spezifikationen, mit denen verteilte reaktive Systemen beschrieben werden. Dabei werden die spezifizierten Systeme mit der Modellierungssprache *Scenario Modeling Language* (SML) beschrieben. Viele der spezifizierten Systeme erhalten parametrisierte Nachrichten mit großen Integer-Parameterwertebereichen aus ihrer Umwelt. Die Menge der Spezifikationen kann aktuell nicht effizient analysiert werden, da sie unter der Zustandsraumexplosion leidet.

Dieses Problem wurde in der vorliegenden Arbeit in Angriff genommen, indem ein Konzept zur symbolischen Ausführung von SML-Spezifikationen entwickelt wurde. Dadurch können große Mengen an parametrisierten Nachrichten zusammengefasst werden, sodass einer Zustandsraumexplosion vorgebeugt wird. Zudem können bei der symbolischen Analyse Pfade zu Fehlern erstellt werden, die konkrete Werte enthalten. Dieses Konzept wurde in einem Prototypen umgesetzt und in ScenarioTools integriert. Dabei wurden verschiedene Verfahren zur Vereinigung von Zuständen entwickelt. Für das Lösen der Gleichungen, die bei der symbolischen Ausführung entstehen, wurde der Z3 SMT-Solver genutzt.

Die Implementierung wurde anhand von Beispielen evaluiert und mit der expliziten Ausführung verglichen. Bei der Evaluation wurden einfache und komplexe Spezifikationen ausgeführt und dabei die Größe des Zustandsraums sowie die benötigte Zeit verglichen. Die Größe des erkundeten Zustandsraums war bei der symbolischen Ausführung wie gewünscht deutlich geringer als bei der expliziten. Darüber hinaus zeigte die symbolische Ausführung bemerkenswerte Performancevorteile, indem sie Zustandsräume in Sekunden erkundete, die von der expliziten Ausführung in zehn Minuten nicht erkundet werden konnten.

Abstract

ScenarioTools is a program for creating and analyzing scenario-based specifications for distributed reactive systems. The specified systems are described with the modeling language *Scenario Modeling Language* (SML). Many of the specified systems receive parameterized messages with large integer parameter value ranges from their environment. The set of these specifications can not be analyzed efficiently, because it suffers from the state space explosion.

This issue was tackled in the present work by the development of a concept for the symbolic execution of SML specification. However, large amounts of parameterized messages can be summarized and thus the state space explosion can be prevented. In addition, paths to errors can be created in the symbolic analysis. These paths contain concrete values. This concept has been implemented in a prototype and integrated into ScenarioTools. Various methods for combining states have been developed. For solving the equations that arise in the symbolic execution, the Z3 SMT solver was used.

The implementation was evaluated by means of examples and compared to the explicit execution. For the evaluation simple and complex specifications were executed and compared in size of the state space and the time required. The explored size of the state space was significantly lower in the symbolic execution than in the explicit. In addition, the symbolic execution showed remarkable performance advantages by exploring state spaces in seconds, that could not be explored by the explicit execution in ten minutes.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **5**
2.1 Symbolische Ausführung 5
2.2 SMT-Solver 8
2.2.1 Allgemein 8
2.2.2 Aufbau 9
2.2.3 SAT-Solver 10
2.2.4 Theorien und Anwendungsgebiete 11
2.3 Scenario Modeling Language 12
2.3.1 Szenariobasierte Spezifikation im Allgemein 12
2.3.2 Beispiel 13
2.3.3 SML-Spezifikation und Objektsystem 13
2.3.4 Kollaborationen, Rollen und Szenarien 15
2.3.5 Nachrichten, Parameter und Runs 16
2.3.6 Aktivierte Nachricht und Ausführungszustand eines Szenarios 17
2.3.7 Unifizierung 17
2.3.8 Alternativ- und Parallel-Konstrukt 17
2.3.9 Interrupt und Violation Bedingung 18
2.3.10 Ausführung und Ausführungszustand 18
2.3.11 Statische und dynamische Rollenbindungen 20
2.3.12 Wildcard-Parameter und Bind-to-Parameter 20
2.3.13 Assumption-Szenario 21
2.3.14 Parameterwertebereiche 22
2.3.15 Verletzungen 23
2.3.16 Sammeln von ausführbaren Nachrichten 23
2.3.17 Szenario-Constraints 24
2.3.18 While und wait-until Konstrukt 25
2.4 ScenarioTools 26
2.5 Zustandsvereinigung bei symbolischer Ausführung 29

3 Anforderungsanalyse **31**
3.1 Problemanalyse . . . 31
3.2 Lösungsansatz . . . 33
3.3 Vision . . . 34
3.4 Anforderungen . . . 35

4 Konzept **37**
4.1 Symbolischer Interpreter . . . 37
4.2 Symbolische Nachrichten . . . 38
4.3 Symbolische Objektattribute . . . 38
4.4 Symbolische Szenariovariablen . . . 39
4.5 Zustands-Constraints . . . 39
4.6 Symbolische Parameterunifizierung und Splitbedingung . . . 40
4.7 Szenario-Constraints . . . 41
4.8 Bedingungen . . . 42
4.9 Multi-Split . . . 42
4.10 Parameterwertebereiche . . . 43
4.11 Listen Operationen . . . 44
4.12 Symbolischer Zustand . . . 47
4.12.1 Zustandsvereinigung mit Teilmengenvergleich . . . 47
4.12.2 Zustandsabstraktion . . . 50
4.12.3 Keine symbolische Zustandsvereinigung . . . 53
4.13 Konkrete Testfälle aus symbolischer Analyse . . . 54
4.14 Anwendungsszenario . . . 55
4.15 Explorationsalgorithmus . . . 56
4.16 Vereinigung von Nachrichten . . . 57
4.17 Modellierung und Analyse von Zeit . . . 57

5 Implementierung **61**
5.1 Neue Plugins . . . 61
5.2 Wichtige Erweiterungen von Metamodellen . . . 62
5.2.1 Konfiguration . . . 62
5.2.2 Symbolische Ausführungslogik . . . 62
5.2.3 Constraints und symbolische Variablen . . . 63
5.2.4 Symbolische Nachrichten . . . 63
5.3 Auswahl des SMT-Solvers für den symbolischen Interpreter . . . 65
5.4 Aufgabenbereiche des symbolischen Interpreters . . . 66
5.5 Zustandsrepräsentation im Zustandsgraphen . . . 67
5.6 Erweiterung der Benutzeroberfläche . . . 68

6 Evaluation **71**
6.1 Tunnel Beispiel . . . 72
6.2 Ofen Beispiel . . . 74
6.3 Kaskadierendes Beispiel . . . 78

6.4 Nicht-lineare Integer-Arithmetik 82

7 Verwandte Arbeiten **85**

8 Zusammenfassung und Ausblick **87**
8.1 Zusammenfassung 87
8.2 Ausblick 89

A Anhang **97**
A.1 Ofensteuerung 97
A.2 Tunnelsteuerung 99
A.3 Kaskade Variante A 102
A.4 Inhalt der DVD 102

Kapitel 1

Einleitung

In immer mehr Bereichen stoßen wir auf softwaregesteuerte Systeme. Viele dieser Systeme bestehen aus reaktiven Komponenten, die untereinander und mit ihrer Umwelt kommunizieren, um gemeinsam komplexe Aufgaben zu erfüllen. Die Einsatzgebiete dieser Systeme liegen immer öfter in sicherheitskritischen Bereichen. Beispiele sind verschiedene Fahrerassistenzsystemen in einem Auto und die Steuerung von Ampelsystemen. Aus dieser Tatsache heraus ist es sehr wichtig, dass diese Systeme frei von Fehlern sind. Zudem kann es sehr teuer werden, wenn Fehler erst in Prototypen oder sogar erst in fertigen Systemen erkannt werden.

Bei der Entwicklung wird das Verhalten der Systeme oft mittels informellen textuellen Anforderungen spezifiziert. Die Überprüfung der Anforderungen auf Konsistenz und Realisierbarkeit geschieht oftmals mittels eines Review-Prozesses. Hierbei ist jedoch keine formelle Überprüfung möglich. Die Ergebnisse des Reviews beruhen lediglich auf Textverständnis und Erfahrungswerten. Dadurch können logische Fehler unentdeckt bleiben, die bei sicherheitskritischen Systemen nicht auftreten dürfen. Darum ist es besser die Anforderungen in einer formellen, überprüfbaren Form darzustellen.

Die textuellen Anforderungen ähneln sehr stak der Beschreibung von Szenarien, wobei beschrieben wird, was ein System in einer bestimmten Situation tun muss oder nicht tun darf. Diese Anforderungen können leicht in formelle Szenarien überführt werden. Dazu bietet sich ScenarioTools[1] mit der szenariobasierten Modellierungssprache *Scenario Modeling Language* (SML) [23, 22, 24] an. ScenarioTools ist ein Programm zur Erstellung und Analyse von szenariobasierten Spezifikationen für verteilte reaktive Systemen. Die Beschreibung der formellen Szenarien in SML ist immer noch sehr ähnlich zu den informellen textuellen Anforderungen. In ScenarioTools gibt es eine Ausführungslogik mit der formelle Szenarien ausgeführt und analysiert werden können. Diese Ausführung basiert auf dem Play-Out-Algorithmus von Harel und Marelly [27], welcher wiederum mit

[1] http://scenariotools.org/

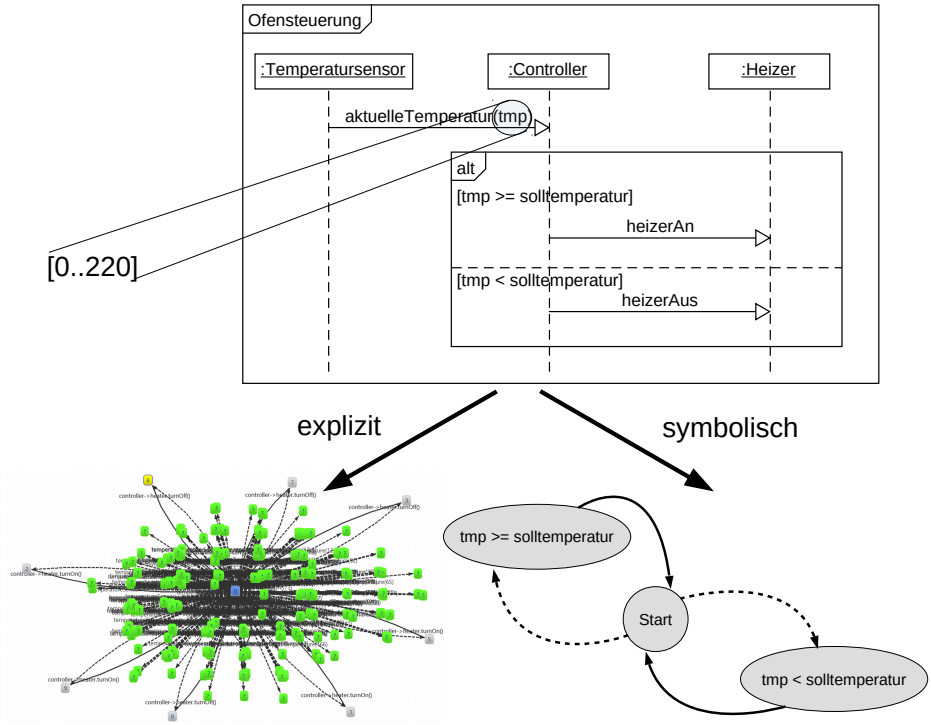

Abbildung 1.1: Vergleich zwischen expliziter und symbolischer Ausführung von einer Spezifikationen mit großen Parameterwertebereiche

Live Sequence Charts (LSC) arbeitet. LSCs stellen eine szenariobasierte Modellierungssprache dar, auf dessen Grundlage SML entwickelt wurde. Während der Ausführung entsteht ein Zustandsgraph, in dem die einzelnen Ausführungszustände des Systems abgebildet werden. Diese Zustände können auf logische Fehler in den Anforderungen untersucht werden. Auf diese Weise können sicherheitskritische Fehler erkannt und somit vermieden werden.

Ab einer gewissen Größe und Komplexität der Spezifikation kann der Zustandsgraph extrem groß werden. Dies ist besonders dann der Fall, wenn die Eingaben aus der Umwelt parametrisiert sind und diese Parameter einen großen Wertebereich besitzen. Dies hängt damit zusammen, dass jeder Eingabeparameter, der vom System verarbeitet wird, einen eigenen Zustand erzeugt. Das kann dazu führen, dass eine effiziente Erstellung des Zustandsgraphen nicht mehr möglich ist. Dieses Phänomen wird Zustandsraumexplosion genannt und verhindert eine vollständige Untersuchung der Zustände auf Fehler.

Das Problem der Zustandsraumexplosion tritt auch schon bei wenig komplexen Spezifikationen mit großen Parameterwertebereichen auf. Zum

Beispiel lässt sich folgender Teil einer Ofensteuerung durch ein einzelnes Szenario beschreiben (siehe Abbildung 1.1). Der Ofen soll bis zu einer gegebenen Solltemperatur aufgeheizt werden. Sobald die Solltemperatur überschritten ist, soll das Heizen ausgestellt werden. Die Größe des Zustandsraumes für dieses einfache System wächst nun mit der Größe des Bereiches, in dem Temperaturen gemessen werden können. Zudem ist sie abhängig von der gegebenen Solltemperatur. Bei einem messbaren Temperaturbereich von 1 bis 100 Grad und 100 verschiedenen Solltemperaturen ergeben sich bereits 10.000 Zustände.

Um diese Abhängigkeit zwischen Parameterwertebereich und Größe des Zustandsgraphs zu verringern, müssen die Parameterwerte auf eine abstrakte Weise betrachten werden. Eine Möglichkeit ist es, die Parameter *symbolisch* zu betrachten [31, 14]. Das heißt es werden keine konkreten Zahlenwerte als Eingabe für Programme verwendet, sondern nicht initialisierte Variablen. Im falle der Ofensteuerung bedeutet dies, wenn sie mit einem symbolischen Parameter ausgeführt wird, dass es nur noch 2 Ausführungspfade gibt, einmal ist der Wert kleiner als die Solltemperatur und einmal ist sie größer gleich der Solltemperatur. Somit haben wir nur noch (zusammen mit dem Startzustand) drei Zustände anstelle der 10.000 expliziten.

In dieser Arbeit wurde solch eine symbolische Ausführung von Integer-Werten für ScenarioTools entwickelt. Dies ermöglicht die formale Analyse für eine größere Klasse von Systemen, die wir auch oft in der Praxis vorfinden. Dabei handelt es sich beispielsweise um Systeme wie die Ofensteuerung, die die Eingaben von Temperatursensoren oder ähnlichen Sensoren verarbeiten. Dies ist in Abbildung 1.1 durch den Parameter *tmp* mit dem Wertebereich von 0 bis 220 dargestellt. Zudem können durch die symbolische Betrachtung von diskreter Zeit, Systeme mit harten Zeitanforderungen formal analysiert werden. Dadurch können unmögliche Zeitanforderungen frühzeitig erkannt werden, die sonst erst in Prototypen erkannt würden. Mit der erweiterten Funktionalität können Eingaben aus der Umwelt als symbolisch definieren werden. Dadurch entstehen einzelne symbolische Zustände, die zahlreiche konkrete Zustände repräsentieren. Das Erzeugen eines Zustandsgraphen aus einer symbolischen Spezifikation führt somit zu einem wesentlich kleineren Zustandsgraphen als es bei einer expliziten Spezifikation der Fall ist. Dennoch sind alle erreichbaren Zustände im Graphen enthalten. Dies ist im unteren Teil der Abbildung 1.1 durch die Gegenüberstellung von expliziten und symbolischen Zustandsgraph dargestellt.

Wird während der Erzeugung des symbolischen Zustandsgraphen ein Fehler gefunden, so liegt dieser in allen dazugehörigen expliziten Fällen vor. Dieser wird wie in der expliziten Form von ScenarioTools durch einen roten knoten im Zustandsgraphen visualisiert. Da der symbolische Pfad zu dem Fehler sehr abstrakt sein kann, wird dieser in einen expliziten Pfad transformiert um die Analyse zu erleichtern. Mit den gewonnen Informationen kann der Spezifikateur diesen Fehler ausbessern und die

Spezifikation erneut testen. Dieser Prozess kann iterativ fortgesetzt werden bis die Spezifikationen keine Fehler mehr enthält. Auf diese Weise können alle Widersprüche auf der Spezifikation entfernt werden, bevor der Prototyp erstellt wird.

Zusammenfassend sind die Ergebnisse meiner Arbeit folgende: Zunächst wurde ein Konzept für die symbolische Ausführung von SML-Spezifikationen entwickelt. Dieses wurde anschließend in einer prototypischen Erweiterung von ScenarioTools implementiert. Dabei wurden verschiedene Verfahren der Zustandsvereinigung in den Prototypen integriert. Darauf folgend wurde eine Evaluation anhand von verschiedenen Beispielen durchgeführt. Das Ergebnis zeigt eine enorme Zustandsreduzierung bei nahezu sämtlichen Spezifikationen mit Integer-Parametern.

Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert: In Kapitel 2 werden die Grundlagen zu szenariobasierten Spezifikationen geklärt und die traditionelle symbolische Ausführung gezeigt. Zudem gibt es einen Einstieg zu SMT-Solvern sowie zu Zustandsvereinigung bei symbolischer Ausführung. Die Anforderungsanalyse in Kapitel 3 beschreibt die Problemstellung und diskutiert Lösungsansätze. Das Konzept für eine Umsetzung der symbolischen Ausführung wird in Kapitel 4 erklärt. Die wichtigsten Elemente der Implementierung werden in Kapitel 5 erläutert. In Kapitel 6 wird das Konzept anschließend evaluiert. Verwandte Arbeiten werden in Kapitel 7 vorgestellt. Zum Schluss folgt in Kapitel 8 eine Zusammenfassung mit Ausblick.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für diese Arbeit relevanten Grundlagen erklärt. Dieses Kapitel gliedert sich wie folgt: In Abschnitt 2.1 wird die grundsätzliche Idee der Symbolischen Ausführung beschrieben. Im zweiten Abschnitt 2.2 werden die Grundlagen zu SMT-Solvern beschrieben. Danach in Abschnitt 2.3 werden die in dieser Arbeit verwendete Modellierungssprache eingeführt und ihr Kontext geklärt. Im letzten Abschnitt wird ein kurzer Einblick in die verschiedenen Techniken der Zustandsverschmelzung gegeben.

2.1 Symbolische Ausführung

Die Idee der symbolischen Ausführung wurde bereits das erste Mal vor 40 Jahren von King [31] und Clarke [14] beschrieben. Dabei ist die Grundidee, dass die Eingaben für Programme oder Funktionen nicht aus konkreten Werten bestehen, sondern diese als nicht initialisierte Variablen betrachtet werden. Die Werte dieser Variablen werden im Laufe der Programmausführung durch Einschränkungen präzisiert. Dies geschieht z.B. dann, wenn die Variable eine Bedingung erfüllen muss. Die daraus resultierenden Einschränkungen werden in den *Pfad-Constraints* gespeichert. Die Pfad-Constraints beschreiben alle möglichen Belegungen der symbolischen Variablen in einem bestimmten Zustand.

Allgemein ist ein *Constraint* eine mathematische Einschränkung für den Lösungsraum einer oder mehrerer Variablen. Zum Beispiel wird mit dem Constraint $x > 5$ der Lösungsraum der Variable x auf Werte größer 5 beschränkt.

Ein symbolischer Zustand ist definiert durch einem Ausführungszustand, den dazugehörigen Pfad-Constraints und einer Menge von symbolischen Variablen. Damit wird eine Menge von konkreten Zuständen beschrieben, bei denen die Wertebelegung der symbolischen Variablen die Pfad-Constraints erfüllen.

Das hier verwendete Beispiel ist von Păsăreanu et al. [40] übernommen.

Das Listing 2.1 zeigt eine Funktion im Java Syntax. Die Funktion berechnet die absolute Differenz zwischen zwei Eingabeparametern. Vor der Rückgabe des Wertes, wird überprüft ob der Wert echt größer Null ist.

In diesem Beispiel wird der Ausführungszustand durch die Zeilennummer in der sich die Ausführung befindet beschrieben. Der Stack sowie der Heap müssen an dieser Stelle nicht betrachtet werden, da das Beispiel so gewählt ist, dass sie keine Auswirkung haben.

```
-1  int absoluteDifference(int x, int y) {
 0    int result;
 1    if (y < x)
 2      result = x - y;
 3    else
 4      result = y - x;
 5    assert result > 0;
 6    return result;
 7  }
```

Listing 2.1: Absolute Differenz

Die symbolische Ausführung des Programms kann als *symbolischer Ausführungsbaum* dargestellt werden. Dieser beschreibt alle möglichen Pfade, die während einer regulären Ausführung erreichbar sind. Bei der konkreten Ausführung der Funktion ist das Ergebnis hingegen ein einzelner Pfad.

In Abbildung 2.1 ist der symbolische Ausführungsbaum der Funktion *absoluteDifference* aus Listing 2.1 zu sehen. Der oberste Zustand (1) ist der Initialzustand. Hier werden die Variablen x und y als symbolische Variablen gekennzeichnet. Die Pfad-Constraints (in der Abbildung mit PC abgekürzt) werden als *wahr* initialisiert. Dies bedeutet, dass es keine Einschränkungen gibt, bzw. dass sie bei beliebiger Variablenbelegung erfüllt sind. Die Transitionen zwischen den Zuständen beschreiben die Ausführung einer Zeile des Quelltextes.

Bei der Ausführung der ersten Anweisung müssen zwei Fälle berücksichtigt werden. Die *if*-Bedingung kann zu *wahr* oder *falsch* evaluiert werden oder zu *wahr* und *falsch*. Diese Entscheidung wird immer unter Berücksichtigung der Pfad-Constraints getroffen. Dabei wird geprüft, ob die Pfad-Constraints *und* die Bedingung gleichzeitig erfüllbar sind, also ob es eine Belegung gibt, die die Formel erfüllt. Genauso wird verfahren um zu zeigen, dass die Bedingung nicht erfüllbar ist und somit der *else*-Fall genommen wird. Dazu werden die Pfad-Constraints und die negierte Bedingung auf Erfüllbarkeit geprüft. Gibt es auch hier eine gültige Belegung, so ist gezeigt, dass der *else*-Fall eintreten kann. Im Beispiel ist der Pfad noch nicht eingeschränkt und beide Fälle sind möglich. Damit ergeben sich zwei neue Zustände. Zustand (2) folgt dem *then*-Fall. Für diesen Zustand werden die Pfad-Constraints mit der Bedingung $sym_x > sym_y$ aus der *if*-Anweisung erweitert. Für den zweiten Zustand (3) werden die Pfad-Constraints mit der negierten Bedingung $\neg(sym_x > sym_y) \Rightarrow sym_x \leq sym_y$ erweitert.

Im *then*-Fall der *if*-Bedingung wird der Variable *result* der Wert $sym_x - sym_y$ zugewiesen (Zustand 4). Damit übernimmt diese Variable die

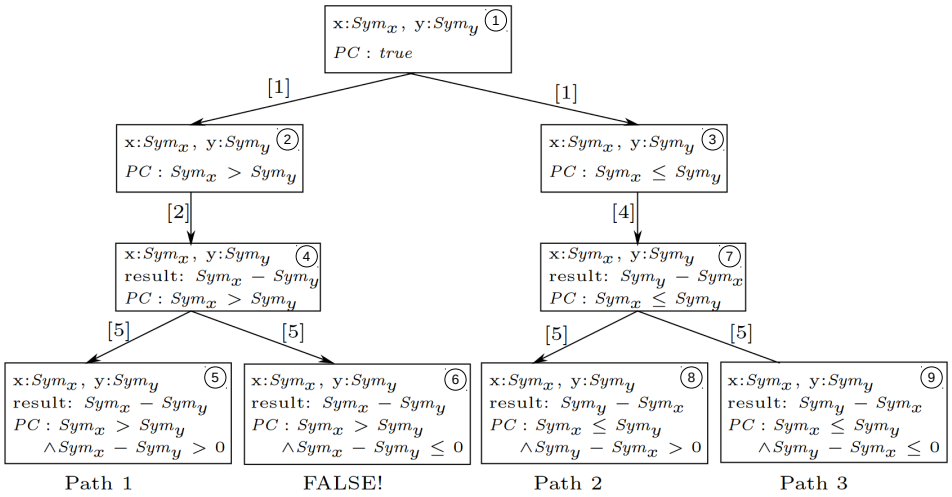

Abbildung 2.1: Symbolischer Ausführungsbaum der Funktion *absoluteDifference* aus Listing 2.1 entnommen aus Păsăreanu et al. [40]

Eigenschaften einer symbolischen Variable. Es folgt die assert-Anweisung in Zeile 5. An dieser Stelle entstehen zwei neue Zustände. Im ersten Fall werden die Pfad-Constraints des aktuellen Zustands um die Bedingung $sym_x - sym_y > 0$ der *assert*-Anweisung erweitert. Im zweiten Fall werden die Pfad-Constraints um die negierte Bedingung $sym_x - sym_y \leq 0$ der *assert*-Anweisung erweitert. Bei der Prüfung der Pfad-Constraints für Zustand 6 stellt sich heraus, dass diese nicht erfüllbar sind. Dies bedeutet, dass dieser Zustand über diesen Pfad nicht erreichbar ist. Auf der anderen Seite des symbolischen Ausführungsbaumes wird der Variablen *result* in Zustand 7 der Wert $sym_y - sym_x$ zugewiesen. Anschließend wird auch hier die *assert*-Anweisung überprüft. Diese ist in beiden Fällen erfüllbar. Damit ist gezeigt, dass das Programm drei gültige Ausführungspfade besitzt, wobei Pfad 3 einen Ausnahmefehler besitzt.

Auf diese Weise können Programme ausgeführt werden ohne ihren Variablen konkrete Werte zuzuweisen. Der symbolische Ausführungsbaum deckt alle Pfade ab, die während der Ausführung durch das Programm führen können. Auf diese Weise kann eine minimale Anzahl an Testfällen erstellt werden, die alle Programmpfade abdecken. Bei beispielsweise einer zufälligen Erstellung von Testfälle liegt die Wahrscheinlichkeit einen Testfall zu wählen, der den Fehler aus dem Beispiel auslöst, bei $5,44^{-7}\%$.

2.2 SMT-Solver

SMT-Solver werden in dieser Arbeit zum Lösen von Pfad-Constraints eingesetzt. Dabei ist ein Solver ein Computer Programm, das mathematische Probleme löst. Entgegen der Meinung von King (1976): „Die symbolische Ausführung von *if*-Anweisungen erfordert Theorembeweise, welche, selbst für modernste Programmiersprachen, mechanisch unmöglich sind.“ [31, Seite 2] sieht dies heutzutage anders aus. In den letzten Jahren haben mathematische Solver große Fortschritte gemacht. SMT-Solver weisen dabei momentan das größte Potential auf. SMT steht für *Satisfiability modulo theories*. Dabei bezieht sich *Satisfiability* auf den Term der Erfüllbarkeit aus der mathematischen Logik. Mit *modulo theories* soll ausgedrückt werden, dass verschiedene Theorien wie Module ausgetauscht und erweitert werden können. Die Theorien stellen dabei Lösungsansätze für Problemklassen, wie beispielsweise lineare Integer-Arithmetik oder Quantoren, dar.

2.2.1 Allgemein

Es gibt viele komplexe und umfangreiche Probleme die sich mit einem SMT-Solver lösen lassen, bei denen andere Solver an ihre Grenzen stoßen. In diese Kategorie fallen zum Beispiel die Stundenplanerstellung oder die Kauf- und Verkaufsentscheidungen von Aktien im Hochfrequenzhandel [6]. Die Überprüfung der Korrektheit von Software fällt ebenfalls in diesen Bereich. Um ein Problem von einem SMT-Solver lösen zu lassen, muss das Problem auf eine mathematische Weise beschrieben werden. Dazu wird das Problem in Form eines Constraint-Systems mathematisch modelliert. Dieses Constraint-System besteht aus einer Menge von Constraints und Behauptungen für eine Menge an Variablen, die gleichzeitig erfüllt sein müssen. Auf diese Weise wird ein Lösungsraum beschrieben. Die Constraint-Systeme können dabei aus hunderten Variablen und tausenden von Gleichungen bestehen.

Diese Menge an Behauptungen und Constraints kann anschließend mit der Frage, ob es unter diesen Bedingungen eine Lösung geben kann, an den SMT-Solver übergeben werden. Ausgehend von der Eingabe sucht der Solver nach einer Lösung. Findet er eine Lösung, so gibt er *SAT* für satisfiable (engl: erfüllbar) zurück. Zusätzlich bietet der Solver für alle Variablen eine konkrete Wertebelegung an, die die Problemstellung erfüllen. Dies wird auch *Model* genannt. Findet der Solver keine Lösung antwortet er mit *UNSAT* für unsatisfiable (engl: nicht erfüllbar).

Einen großen Vorteil erlangen die SMT-Solver dadurch, dass sie noch eine dritte Möglichkeit haben zu antworten: *UNKNOWN* (engl: unbekannt). Damit kann der SMT-Solver mitteilen, dass er dieses Problem nicht lösen kann. Dies ist besonders wichtig, da viele Probleme, die ein SMT-Solver lösen soll, im Bereich der NP-Vollständigen Probleme liegen bis hin zu Problemen die generell unentscheidbar sind.

Der Grund weshalb sich viele Probleme trotz der hohen Komplexität lösen lassen ist, dass keine zufällig generierten Probleme gelöst werden. Diese Probleme bringen eine gewisse Struktur mit sich. Diese Strukturen versucht der Solver zu erkennen. Dies ist dadurch möglich, da die Variablen in den Gleichungen eine große Abhängigkeit zueinander besitzen. Dadurch werden die Freiheitsgrade der Variablen sehr stark eingeschränkt und das Problem somit einfacher. Auf Grund dieser Tatsache kann im besten Fall eine Antwort in Sekundenbruchteilen gegeben werden. Im schlechtesten Fall findet der Solver jedoch *nie* eine Antwort bzw. antwortet mit einem *TIMEOUT* oder *UNKNOWN*.

2.2.2 Aufbau

SMT-Solver bestehen im Kern aus zwei Hauptkomponenten [15, 16]. Die erste Hauptkomponente ist ein SAT-Solver (siehe Abschnitt 2.2.3). Dieser kann Bool'sche Formeln auf Erfüllbarkeit prüfen. Da SMT-Solver in der Regel mit komplexeren Formeln arbeiten als mit Bool'schen Formeln, müssen die Eingaben im ersten Schritt darauf reduziert werden. Somit würde die Formel von λ auf ψ reduziert werden.

$$\lambda = x > 5 \wedge y \leq 5 \wedge (y = x \vee y < 3)$$

$$\Rightarrow \psi = F1 \wedge F2 \wedge (F3 \vee F4)$$

Die substituierten Teilformeln werden nun an einen Theorie Solver übergeben, welcher die zweite Hauptkomponente eines SMT-Solvers darstellt. Eine weitere Komponente stellt eine übergeordnete Einheit dar, die passende Theorien zu der gegebenen Formel auswählt. Und darin liegt eine der Stärken der SMT-Solver. Sie können eine Vielzahl an Theorien unterstützen, die jeweils untereinander vermischt werden können (siehe Abschnitt 2.2.4).

Für das oben gegeben Beispiel würde die Theorie der linearen Integer-Arithmetik gewählt werden. Anschließend sucht der SAT-Solver nach einer Belegung für die vereinfachte Formel und gibt diese an den domänenspezifischen Solver weiter. Dieser weiß nun welche der substituierten Teilformeln zusammen erfüllt sein müssen.

In diesem Fall wäre eine gültige Belegung für ψ die folgende: $F1, F2, F3$. Der Theorie Solver sucht nun nach einer Belegung für $x > 5, y \leq 5, y = x$. Für diese Constraints gibt es allerdings keine gültige Lösung. Nun sucht der SAT-Solver nach einer weiteren gültigen Belegung für ψ und findet folgende: $F1, F2, F4$. Diese übergibt er wieder dem Theorie Solver. Für die nun zu prüfenden Constraints $x > 5, y \leq 5, y < 3$ gibt es eine Lösung. Daraus folgt, dass das Constraint-System erfüllbar ist. Zudem erstellt der SMT-Solver ein Modell für λ mit einer Variablenbelegung, die das Constraint-System

erfüllt: $M = \{x = 6, y = 2\}$.

2.2.3 SAT-Solver

Moderne SAT-Solver wie beispielsweise DPLL(T) basierte Solver verwenden *konfliktgetriebenes Klausel-Lernen*(CDCL)[15, 16]. Sie verwenden Formeln in *konjunktiver Normalform* (KNF) als Eingabe. Dabei handelt es sich um oder-Verknüpfte Literale, die als Klauseln zusammen gefasst werden. Ein Literal ist atomar und kann den Wert 0 oder 1, bzw. wahr oder falsch annehmen. Diese Klauseln werden wiederum und-Verknüpft. Jede aussagenlogische Formel kann in KNF umgeformt werden. Folgendes Beispiel zeigt eine Formel in KNF:

$$\varphi = \neg x_2 \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Um eine solche Formel zu losen führt ein SAT-Solver als ersten Schritt eine *Unit Propagation* durch. Dabei wird nach Klauseln gesucht, die nur ein Literal enthalten, auch *Unit Klausel* genannt. Dieses Literal muss so belegt werden, dass die Klausel erfüllt ist. Wird eine Klausel nicht erfüllt, so ist die gesamte Formel nicht erfüllt. In diesem Fall wird $x_2 = 0$ gesetzt, um die Klausel zu erfüllen. Alle Entscheidungen werden in einem Implikationsgraphen festgehalten. Anschließend kann die Formel wie folgt vereinfacht werden:

$$\varphi = (x_1 \vee \neg x_3) \wedge (x_1 \vee x_3)$$

Gibt es keine *Unit Klausel*, wird ein Literal zufällig bestimmt. Dies ist in unserem Beispiel der Fall. Wählen wir zufällig die Belegung $x_1 = 0$, vereinfacht sich die Formel wie folgt:

$$\varphi = (\neg x_3) \wedge (x_3)$$

Nun kann wieder Unit Propagation ausgeführt werden. Dabei kommt es zu einem Konflikt, da die Belegung einen Widerspruch erzeugt wird. x_3 kann nicht gleichzeitig wahr und falsch sein.

Wird eine Belegung durch eine Unit Klausel erzwungen oder durch eine zufällig Wahl bestimmt, die zur nicht Erfüllung der Gleichung führt, müssen die Belegungen, die im Konflikt stehen, zurück genommen werden. Die Literal Belegungen, die zurückgenommen werden müssen, lassen sich aus dem Implikationsgraphen ablesen. In unserem Fall ist es x_1.

Nachdem die Belegung zurückgenommen wurde kann eine neue Klausel gelernt werden. Für das Beispiel sieht die gelernte Klausel wie folgt aus:

$$\neg(\neg x_2 \wedge \neg x_1) \Rightarrow x_1 \vee x_2$$

Die gelernte Klausel wird der Formel von φ hinzugefügt und bereits belegte Variablen können direkt gekürzt werden. Mit der neu gelernten Klausel wird

der zweite Schritt nun durch eine Unit Klausel impliziert und muss nicht mehr zufällig gewählt werden. Damit wird x_1 auf wahr gesetzt und die übrigen Klauseln sind erfüllt.

$$\varphi = (x_1 \vee \neg x_3) \wedge (x_1 \vee x_3) \wedge (x_1 \vee \not{x_2})$$

Die resultierenden Belegung $x_1, \neg x_2$ erfüllt die Formel.

Aktuelle SAT-Solver optimieren diese Verfahren und erweitern es beispielsweise um nicht-synchrones Backtracking, Suchheuristiken und verbessertes Klausel-Lernen. Weiteres findet sich unter Marques-Silva et al. [37].

2.2.4 Theorien und Anwendungsgebiete

Es gibt viele verschiedene Theorien, die von SMT-Solvern unterstützt werden. Einiger der wichtigsten Theorien sind in diesem Abschnitt kurz erläutert. Ein Überblick über die wichtigen Theorien findet sich unter Solar-Lezama [46]. Eine Liste von standardisierten Theorien und Logiken werden von der *SMT-Lib* [5] zur Verfügung gestellt. Allgemein gefasst sind viele der Theorien dazu ausgelegt, die Korrektheit von Computerprogrammen zu beweisen.

Lineare Integer-Arithmetik ist auf *Presburger Arithmetik* reduzierbar. Diese wiederum ist definiert durch die Signatur $(0, 1, +, -, \leq)$ (siehe Kapitel 12.2.2.2. in [7]). Diese Theorie unterstützt die Verwendung von Integern. Und ist in polynomieller Zeit lösbar. Die Theorie kann dazu verwendet werden Zeiger Arithmetik zu überprüfen.

Lineare und Nicht-lineare Real-Arithmetik diese Theorien sind entscheidbar.

Nicht-lineare Integer-Arithmetik ist nicht entscheidbar. Dies visualisiert folgendes Beispiel:

$$y = x^2 \mapsto \sqrt{y} = x$$

Wenn mit Reellen Zahlen gearbeitet wird, kann diese Formel einfach umgestellt und die Lösung bestimmt werden, indem die Wurzel aus y gezogen wird. Ein Solver müsste an dieser Stelle einfach eine beliebige Zahl für y einsetzen die größer als 0 ist und könnte die Gleichung lösen. Im Bereich der Integer geht das nicht, da hier nur ganzzahlige Lösungen akzeptiert werden. Es gibt auch Gleichungen die keine ganzzahligen Lösung besitzen, siehe dazu *Diophantische Gleichung* [38].

Bit Vektor Theorie wird eingesetzt, um Überlauffehler in Programmen zu finden.

Array Theorie kann dazu verwendet werden, Kollektionen zu untersuchen, bei denen die Größe erst zur Laufzeit festgelegt wird. Des Weiteren kann

der Heap eines Computers als Array abstrahieren werden. Dabei wird der Index des Arrays als Speicheradresse verwendet. Der Wert repräsentiert den zugehörigen Wert an der Speicheradresse.

Uninterpretierten Funktionen Die Theorie der uninterpretierten Funktionen kann dazu verwendet werden, Funktionen zu abstrahieren, die bei gleicher Eingabe immer die gleiche Ausgabe erzeugen. Die Abstraktion von Funktionen ist zum Beispiel dann sinnvoll, wenn der Aufwand der Modellierung einer Funktion zu groß ist oder die Funktion nicht Gegenstand des Tests ist. Beispiele für solche Funktionen sind zum Beispiel die Sinusfunktion oder der Wurzelfunktion

Quantoren erhöhen die Komplexität für den Solver ungemein. Sie sind wichtig, um beispielsweise Invarianten von Schleifen zu beschreiben.

2.3 Scenario Modeling Language

Die *Scenario Modeling Language* (SML) ist eine textuelle, szenariobasierte Modellierungssprache, die auf *Live Sequence Charts* [27] und *Modal Sequence Diagrams* [9, 26] basiert. SML wurde von Greenyer et al. [23, 22, 24] eingeführt. Mit ScenarioTools existiert eine Ausführungslogik mit der SML-Spezifikationen ausgeführt werden können.

In diesem Abschnitt erfolgt zuerst ein allgemeiner Überblick zu szenariobasierten Spezifikationen. Danach folgt eine Einführung in die für diese Arbeit wichtigen Konzepte von SML und wie diese von der Ausführungslogik umgesetzt werden.

2.3.1 Szenariobasierte Spezifikation im Allgemein

Szenariobasierte Spezifikationen eigenen sich besonders für die Beschreibung der Kommunikation zwischen verschiedenen Elementen eines Systems, die auf Eingaben aus der Umwelt reagieren. Dabei liegt der Austausch von Nachrichten zwischen den einzelnen Komponenten des Systems im Fokus. Beim szenariobasierten Ansatz kann das Verhalten der einzelnen Elemente als Interaktion zwischen den Elementen in Form von Szenarien beschrieben werden. Eine erste Form der szenariobasierten Spezifikation wurde von Harel und Marelly [27] vorgestellt. Dabei werden Szenarien mit *Live Sequenz Charts* (LSC) beschrieben. LSCs sind eine erweiterte Form der UML Sequenzdiagramme. Sie beschreiben was ein System in einer bestimmten Situation tun kann, muss oder nicht tun darf. Dadurch wird eine *liveness* Bedingung beschrieben, welche im Namen festgehalten wurde [27, Seite 16]. Eine szenariobasierte Spezifikation besteht aus einer Menge an einzelnen Szenarien (LSC), die sich während der Ausführung über Nachrichten synchronisieren.

2.3.2 Beispiel

Zur Veranschaulichung der Konzepte von SML wird Schritt für Schritt eine SML-Spezifikation als Beispiel aufgebaut. Das vollständige Beispiel mit zusätzlichen Ergänzungen ist im Anhang A.1 zu finden. Das Beispiel ist eine einfache Ofensteuerung. Ausgehend von einer Solltemperatur und einer gemessenen Temperatur steuert sie, ob der Heizvorgang fortgesetzt oder beendet werden muss. Folgende Anforderungen *(Rx)* und Annahmen *(Ax)* gelten für die Ofensteuerung:

R1 Der Ofen kann an- oder ausgestellt werden.
R2 Der Ofen hat einen Temperatursensor, der dem Controller die aktuelle Temperatur mitteilt.
R3 An einem Bedienpanel kann die Solltemperatur des Ofens eingestellt werden.
R4 Wenn der Ofen angeschaltet ist, signalisiert der Controller mittels einer Vorheizlampe, ob er sich in der Aufheizphase befindet.
R5 Ist die gemessene Temperatur größer oder gleich der Solltemperatur, dann soll der Heizer im Ofen ausgestellt werden.
R6 Ist die gemessene Temperatur kleiner als die Solltemperatur, dann soll der Heizer im Ofen angestellt werden.
A1 Die gemessenen Temperaturen befinden sich in einem Wertebereich von 0 bis 220 Grad Celsius.
A2 Die Solltemperatur hat einen Wertebereich von 0 bis 220 Grad Celsius.
A3 Die Temperaturentwicklung findet in 1 Grad Schritten statt.

2.3.3 SML-Spezifikation und Objektsystem

Eine SML-Spezifikation enthält die Anforderungen und Annahmen an ein System in Form von Szenarien. Die einzelnen Komponenten des Systems und seiner Umwelt werden durch Objekte eines Objektsystems repräsentiert. Diese Objekte tauschen untereinander Nachrichten aus. Dabei werden die Objekte des Objektsystems in Umwelt- und Systemobjekte unterteilt. Systemobjekte sind dabei kontrollierbar, Umweltobjekte hingegen nicht. In Abbildung 2.2 ist das Objektsystem zusehen. Alle Objekte zusammen stellen das beschriebene System dar. Im unteren Teil der Abbildung ist ein Szenario zu sehen, das ein Teil der Spezifikation darstellt. Die Rollen im Szenario repräsentieren Objekte des Objektsystems. Im oberen Teil der Abbildung ist ein Klassendiagramm zu sehen. Das Objektsystem ist eine Instanziierung des Klassendiagramms. Die Klassen des Klassendiagramms haben Operationen, diese stellen die Nachrichten dar, die die Objekte empfangen können. Diese sind auch in der Spezifikation wiederzufinden.

Im Beispiel der Ofensteuerung (Listing 2.2) findet die Auswahl des Klassendiagramms in Zeile 5 statt. Für die Definition wird hier ein *ecore-Klassendiagramm* des ECLIPSE Modeling Frameworks[1] verwendet. In Zeile

[1] https://eclipse.org/modeling/emf

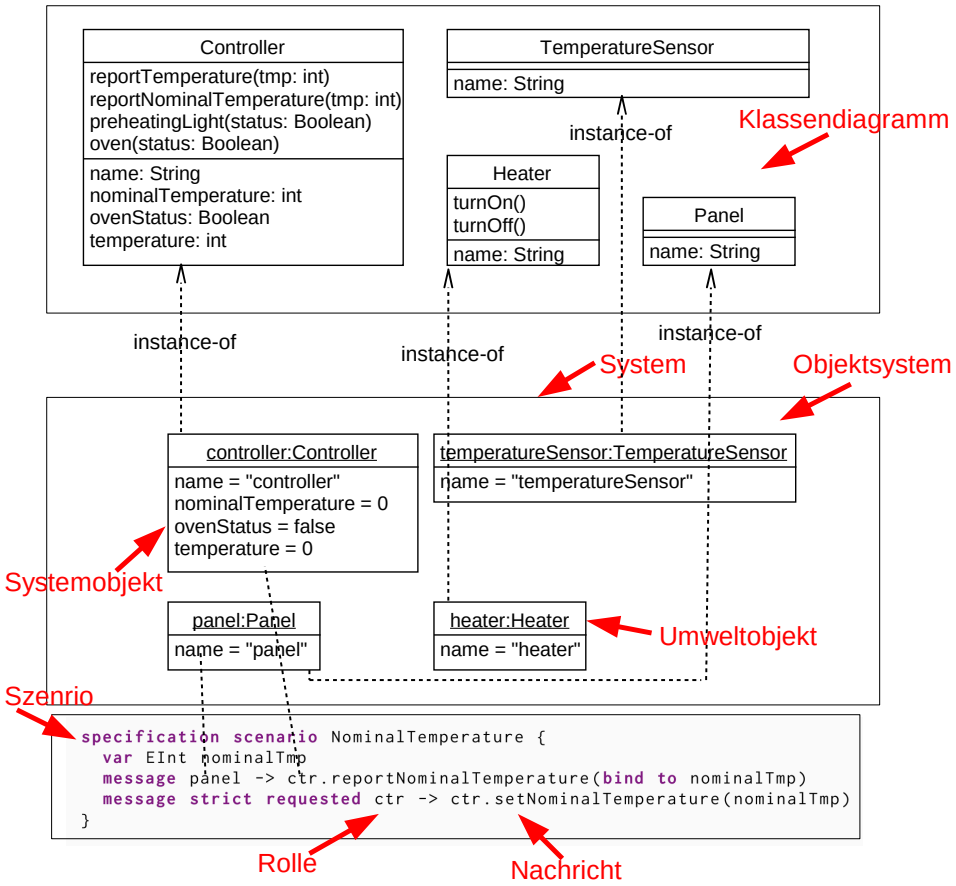

Abbildung 2.2: Das Objektsystem als Instanz eines Klassendiagramms im Zusammenhang mit einem Szenario

7 bis 11 werden die verschiedenen Objekte, die Instanzen der Klassen sind, in zwei Kategorien unterteilt. Die Objektklassen, die als *controllable* gekennzeichnet sind gehören zu den Systemobjekten. Mit *uncontrollable* gekennzeichnete Objektklassen sind hingegen Umweltobjekte. Eine Spezifikation enthält eine oder mehrere *Kollaborationen*.

```
1  import "../model/oven.ecore"
2
3  system specification OvenSpecification {
4
5    domain oven
6
7    define Controller as controllable
8    define TemperatureSensor as uncontrollable
9    define HumiditySensor as uncontrollable
10   define Heater as uncontrollable
11   define Panel as uncontrollable
12
13   collaboration OvenCollaboration { ... }
14 }
```

Listing 2.2: Ofen Spezifikation

2.3.4 Kollaborationen, Rollen und Szenarien

In einer *Kollaboration* wird das Verhalten des Systems beschrieben. Dazu wird zuerst die Menge an benötigen *Rollen* definiert. Dabei repräsentiert eine Rolle ein Objekt des Objektsystems. Rollen können fest an die Objekte gebunden werden, dann handelt es ich um *statische Rollen*. In Listing 2.3 ist die Definition der Rollen in Zeile 2 bis 7 zu sehen. Für die verschiedenen Rolle werde Namen vergeben (wie beispielsweise *ctr*). Zudem wird der Rolle ein Typ aus dem Klassendiagramm zugewiesen. Der Typ der Rolle *ctr* ist *Controller*.

```
1  collaboration OvenCollaboration {
2
3    static role Controller ctr
4    static role TemperatureSensor ts
5    static role HumiditySensor hs
6    static role Heater heater
7    static role Panel panel
8
9    specification scenario OvenRegulation {}
10 }
```

Listing 2.3: Ofen Kollaboration

Darüber hinaus enthalten Kollaborationen Szenarien. In einem Szenario wird das Verhalten verschiedener Rollen in einer bestimmten Situation beschrieben. Szenarien vom Typ *specification* beschreiben Systemverhalten. Abfolgen von Interaktionen zwischen Objekten werden dabei als Nachrichten beschrieben. Das definierte Verhalten in diesem Szenario ist für kontrollierbare Objekte bindend. Szenarien können Variablen besitzen, sog. *Szenariovariablen*. Diese können vom Typ *Integer*, *Boolean*, *String*, *Enum* oder *Objekt* sein.

In Listing 2.4 ist das Szenario *OvenRegulation* zu sehen. Es beschreibt das Verhalten des Systems nachdem der Temperatursensor (*ts*) eine Temperatur gemessen hat und sie dem Controller (*ctr*) mitteilt. Die gemessene Temperatur wird in der Szenariovariable in Zeile 2 zwischengespeichert. Diese ist vom Typ Integer. Anschließend wird geprüft, ob der Ofen eingestellt ist. Darauf folgend speichert der Controller die Temperatur in einem Attribut. Im letzten Block wird entschieden, ob die gemessene Temperatur größer oder kleiner als die Solltemperatur ist und danach entschieden ob die Nachricht zum Ein- oder Ausschalten gesendet werden soll.

```
1  specification scenario OvenRegulation {
2    var EInt temperature
3    message ts -> ctr.reportTemperature(bind to temperature)
4    interrupt if [ controller.ovenStatus == false ]
5    message strict requested ctr -> ctr.setTemperature(temperature)
6    alternative if [temperature > controller.nominalTemperature]{
7      message strict requested controller -> heater.turnOff()
8    } or if [ temperature <= controller.nominalTemperature]{
9      message strict requested controller -> heater.turnOn()
10   }
11 }
```

Listing 2.4: Szenario OvenRegulation

2.3.5 Nachrichten, Parameter und Runs

Eine Nachricht besteht aus einem *sendenden Objekt* und einem *empfangenden Objekt*. Diese sind in der Spezifikation jeweils durch eine Rolle repräsentiert. Eine Nachricht hat einen *Namen* und eine Menge an *Parametern*. Diese Parameter können vom Typ *Integer*, *Boolean*, *String*, *Enum* oder *Objekt* sein. Wird eine Nachricht von einem Umweltobjekt gesendet, heißt sie *Umweltnachricht*. Wird sie von einem Systemobjekt gesendet, heißt sie *Systemnachricht*. Zudem können Nachrichten mit dem Schlüsselwort *strict* gekennzeichnet werden. Dies hat Auswirkungen auf die Behandlung von Verletzungen. Siehe 2.3.15 für eine Erklärung. Besitzen die Nachricht dieses Schlüsselwort nicht, so wird sie als *non-strict* bezeichnet. Mit dem Schlüsselwort *requested* werden vom Szenario geforderte Nachrichten beschrieben. Ist eine Nachricht *requested*, dann *muss* sie im Laufe der Ausführung gesendet werden. Ist die Nachricht nicht *requested*, dann *kann* sie im Laufe der Ausführung auftreten.

Ein Beispiel für eine Umweltnachricht ist in Zeile 3 des Listings 2.4 zu sehen, da der Sender (*ts*) dieser Nachricht in Listing 2.2 als unkontrollierbar definiert wurde. Eine Systemnachricht ist in Zeile 5 des Listings 2.4 zu sehen.

Schreibender Zugriff auf Objekte des Objektsystems kann mit sog. *set-Nachrichten* erfolgen (siehe Zeile 5 Listing 2.4). An dieser Stelle wird das Attribut des Objektes, das an die Rolle des *Controllers* gebunden ist, gleich dem Wert der Szenariovariable *temperature* gesetzt.

Ein *Run* stellt eine unendliche Abfolge von Nachrichten dar, die im Objektsystem gesendet werden. Dabei handelt es sich um eine unendliche

Abfolge, da die betrachteten Systeme unendlich lange laufen. Die gesendeten Abfolgen von Nachrichten werden dabei von den Szenarien akzeptiert. Ein Run bei dem alle Nachrichten einer Abfolge von den Szenarien akzeptiert werden, erfüllen die Spezifikation.

2.3.6 Aktivierte Nachricht und Ausführungszustand eines Szenarios

Nach der Aktivierung eines Szenarios hat dieses einen Ausführungszustand. Dieser verläuft sequenziell durch das Szenario. Die Nachricht, bei der sich die Ausführung befindet, wird *aktivierte Nachricht* genannt. Alle weiteren Nachrichten, die in diesem Szenario vorkommen und nicht aktiviert sind, werden als *lauschende Nachrichten* bezeichnet. Ist die aktivierte Nachricht vom Typ *strict*, wird der Ausführungszustand dieses Szenarios als *strict* bezeichnet. Sonst wird der Zustand des Szenarios als *non-strict* bezeichnet. Listing 2.5 zeigt eine aktivierte Nachricht, die nicht gefordert ist jedoch *strict*.

```
1 specification scenario NominalTemperature {
2 var EInt nominalTmp
3 message panel -> ctr.reportNominalTemperature(bind to nominalTmp)
4  message strict requested ctr -> ctr.setNominalTemperature(nominalTmp)
5 }
```

Listing 2.5: Darstellung einer aktivierten Nachricht im *Play-Out-Modus* (grün hinterlegt)

2.3.7 Unifizierung

Nachrichten können miteinander *unifiziert* werden. Dabei wird eine Nachricht, die zwischen Objekten gesendet wurde, mit Nachrichten in den Szenarien verglichen. Zwei Nachrichten heißen unifizierbar, wenn die Nachrichten den gleichen Empfänger sowie Sender haben und die Signatur der Nachrichten gleich ist. Eine Nachricht heißt *parameterunifizierbar* wenn sie unifizierbar ist, die Parametertypen gleich sind und die Parameterwerte übereinstimmen.

2.3.8 Alternativ- und Parallel-Konstrukt

Eine Alternative beschreibt eine *Entweder-Oder-Weiche* in der Ausführung. Dabei kann eine Alternative einen bis beliebig viele Fälle besitzen. Ein Fall kann an eine Bedingung geknüpft sein, sodass er nur aktiviert werden kann, wenn die Bedingung erfüllt ist. In Zeile 6 Listing 2.4 befindet sich eine Alternative mit 2 Fällen, die jeweils eine Bedingung besitzen, wobei sich die Bedingungen gegenseitig ausschließen. Die gemessene Temperatur ist entweder größer als der Sollwert (dann wird das Heizen beendet), oder die gemessene Temperatur ist kleiner als oder gleich der Solltemperatur (der Heizer wird angestellt).

Ein Parallelkonstrukt erlaubt die parallele Ausführung von zwei oder mehr Blöcken. Dieses Konstrukt kann immer dann eingesetzt werden, wenn die Reihenfolge zweier Nachrichten unwichtig ist. Das Parallel-Konstrukt aus Listing 2.6 beschreibt beispielsweise zwei Nachrichten, die in beliebiger Reihenfolge ausgeführt werden können.

```
1 parallel {
2   message requested sender -> empfaenger.nachricht()
3 } and {
4   message requested sender -> empfaenger.nachricht2()
5 }
```

Listing 2.6: Ausschnitt der ein Parallelkonstrukt zeigt.

2.3.9 Interrupt und Violation Bedingung

Wird ein *Interrupt* Element während der Ausführung erreicht, wird die Bedingung ausgewertet. Ergibt die Auswertung *false*, wird die Ausführung mit dem nächsten Element fortgesetzt. Ergibt die Auswertung *true*, wird das Szenario an dieser Stelle beendet.

Siehe Zeile 4 Listing 2.4. An dieser Stelle wird das Szenario nur weiter ausgeführt, wenn der Ofen an ist. Ist der Ofen aus (ofenStatus ist *false*), dann wird das Szenario an dieser Stelle beendet.

Mit einem *Violation if* Konstrukt kann während der Ausführung eine Sicherheitsverletzung ausgelöst werden. Kann die Bedingung zu *wahr* evaluiert werden, so wird die Ausführung an dieser Stelle beendet. Dies ist eine Situation, in der ein Szenario einen Run nicht akzeptiert. Listing 2.7 zeigt eine solche Bedingung.

```
1 violation if [Bedingung]
```

Listing 2.7: Ausschnitt zeigt eine Bedingung, die eine Sicherheitsverletzung auslösen kann

2.3.10 Ausführung und Ausführungszustand

Um eine SML-Spezifikation im *Play-Out-Modus* ausführen zu können wird ein Objektsystem benötigt. Ein gültiges Objektsystem ist eine Instanziierung des Klassendiagramms, das in der Spezifikation angegeben wird. Das Objektsystem wird in der Konfigurationsdatei mit der Spezifikation verbunden (siehe Listing A.2 im Anhang A.1). Die Spezifikationsdatei wird in Zeile 1 der Konfigurationsdatei geladen. In der darauf folgenden Zeile wird die enthaltene Spezifikation explizit ausgewählt. Das zugehörige Objektsystem wird in Zeile 3 ausgewählt(siehe A.1). In Zeile 5 bis 11 werden die statischen Rollen an Objekte aus dem Objektsystem gebunden. Dies wird auch als Rollenbindung verstanden. Damit kann die Spezifikation ausgeführt werden.

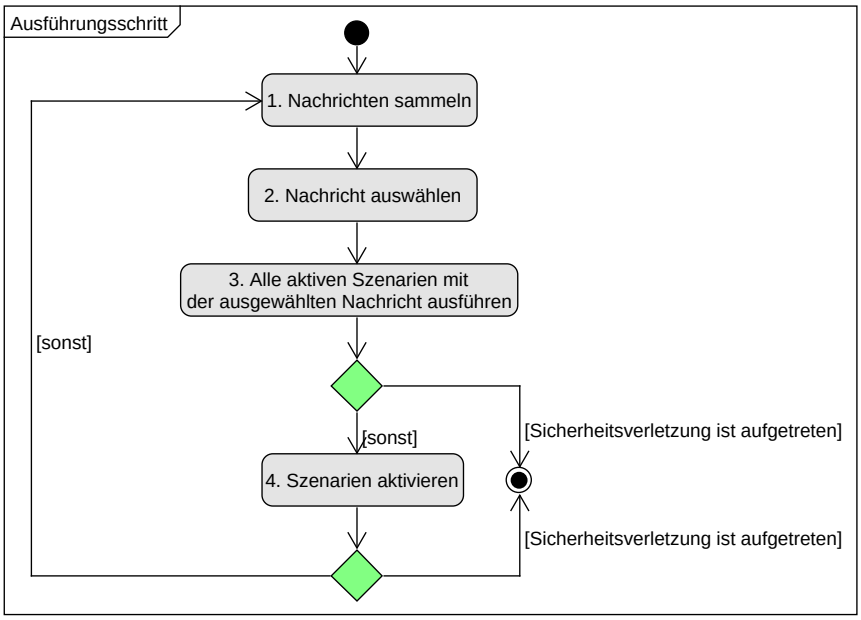

Abbildung 2.3: Ausführung eines Schrittes

Mit einer SML-Spezifikationen werden reaktive Systeme beschrieben. Dabei wartet das System auf eine Eingabe aus der Umwelt: Eine Umweltnachricht. Erfolgt diese, antwortet das System mit einer Abfolge von Systemnachrichten. Bei der Ausführung von SML-Spezifikationen wird die Annahme getroffen, dass das System immer schnell genug ist, um auf eine Umweltnachricht mit beliebig, aber endlich vielen Systemnachrichten zu antworten, bevor die nächste Umweltnachricht gesendet wird.

Die Ausführung einer SML-Spezifikation im *Play-Out-Modus* folgt den Schritten im Aktivitätsdiagramm 2.3. Die Aktivitäten sind eine sehr abstrakte Beschreibung des Vorgangs. Im ersten Schritt werden alle Nachrichten gesammelt, die ausgeführt werden können und einen Einfluss auf das modellierte System haben können. Im zweiten Schritt wird eine dieser Nachrichten ausgewählt. Anschließend werden in Schritt 3 alle aktiven Szenarien mit dieser Nachricht ausgeführt. Wenn es an diesem Punkt eine Verletzung der Spezifikation gibt, wird die Ausführung beendet. Sonst folgt in Schritt 4 die Aktivierung von Szenarien. Dabei wird ein Szenario aktiviert, wenn die erste Nachricht des Szenarios mit der ausgeführten Nachricht unifiziert werden kann (siehe Abschnitt 2.3.7). Tritt an dieser Stelle eine Verletzung der Spezifikation auf, wird die Ausführung ebenso beendet. Ansonsten fährt die Ausführung erneut mit Schritt 1 fort. Die Bedeutung der einzelnen Schritte wird im folgenden Verlauf noch detaillierter erläutert.

Bei der Ausführung von Spezifikationen wird der Ausführungszustand durch den Zustand des Objektsystems und die Menge der Ausführungszustände der Szenarien beschrieben. Dabei wird der Zustand eines Szenarios durch seine Rollenbindungen, die Belegung der Szenariovariablen und durch die aktuell aktivierten Elementen beschrieben.

2.3.11 Statische und dynamische Rollenbindungen

Es gibt Rollen, die statisch gebunden werden. Statische Rollenbindungen gelten die gesamte Ausführung lang und werden von einem Objekt belegt. Statische Rollen werden in der Konfigurationsdatei definiert. Ein Beispiel für statische Rollenbindungen ist in Listing A.2 Zeile 5 bis 11 zu sehen. Dynamische Rollenbindungen werden bei der Initialisierung eines Szenarios gebunden. Dabei werden die ersten beiden Rollen immer vom Sender und dem Empfänger der ersten Nachricht gebunden. Alle weiteren Rollen die in dem Szenario vorhanden sind, können mit Hilfe von Bedingungen an Objekte gebunden werden. Dabei können die Bedingungen auf Attribute der bereits gebundenen Rollen zugreifen.

In Listing 2.8 ist ein generisches Beispiel einer dynamischen Rollenzuweisung zu sehen. Dabei wird der ungebundene Rolle *ungebundeneRolle* der Attributwert der bereits gebundenen Rolle *sender* zugewiesen. In diesem Fall steht *sender* für den Sender der ersten Nachricht. Diese Rolle wird automatisch bei Aktivierung des Szenarios gebunden.

```
1 dynamic role Type ungebundeneRolle
2
3 specification scenario aScenario with dynamic bindings [
4   bind ungebundeneRolle to sender.attribute
5 ]{
6   message sender -> empfaenger.fistMessage()
7   ...
8 }
```

Listing 2.8: Rollenbindung der dynamischen Rolle *ungebundeneRolle*.

2.3.12 Wildcard-Parameter und Bind-to-Parameter

Ist für die Ausführung eines Szenarios nur die Nachricht ohne konkreten Parameterwert entscheidend, so kann an dieser Stelle eine *Wildcard* gesetzt werden (siehe Zeile 3 Listing 2.4). Dies bedeutet, dass die Nachricht mit jedem beliebigen Wert parameterunifizierbar ist. Alle Nachrichten, die von Objekten zu anderen Objekten gesendet werden, haben einen konkreten Wert.

```
1 message temperatureSensor -> controller.reportTemperature(*)
```

Listing 2.9: Nachricht mit Wildcard-Parameter

Der *bind-to-Parameter* verhält sich wie der Wildcard-Parameter. Zusätzlich wird bei der Unifizierung einer aktivierten Nachricht der Wert der gesendeten Nachricht an eine Szenariovariable gebunden.

Das Listings 2.4 zeigt das Szenario *OvenRegulation*. Das Szenario wird aktiviert, sobald eine Nachricht auftritt die mit der ersten Nachricht (Zeile 3) unifiziert werden kann. Bei der Aktivierung des Szenarios wird der Parameter der Nachricht an die Szenariovariable *temperature* gebunden. Im nächsten Schritt wird geprüft, ob der Ofen ausgestellt ist (Zeile 4). Ist dies der Fall wird das Szenario an dieser Stelle beendet. Ist der Ofen an, so wird die Nachricht *setTempreature(temperature)* aktiviert. Da diese vom Szenario gefordert wird (*requested*), wird sie im nächstem Schritt ausgeführt. Anschließend folgt eine Alternative (Zeile 6), bei der die Bedingungen der einzelnen Fälle ausgewertet werden. In diesem Fall schließen sich die Fälle gegenseitig aus. Je nachdem ob die Szenariovariable größer oder kleiner als die Solltemperatur ist, wird als nächstes die Nachricht gefordert, die den Heizer ausstellt bzw. anstellt. Nachdem die geforderte Nachricht ausgeführt wurde, wird das Szenario normal beendet.

2.3.13 Assumption-Szenario

Ein *Assumption-Szenario* beschreibt das Verhalten der Umwelt. Da die Umwelt an sich nicht kontrollierbar ist, werden diese Szenarien als Annahmen an die Umwelt verstanden.

In unserem Beispiel haben wir einen Temperatursensor, der beliebige Werte messen kann. Dadurch werden die Möglichkeiten in jedem Schritt stark erhöht. Diese Tatsache trägt zum Problem der Zustandsraumexplosion bei. Um die Möglichkeiten des Sensors einzugrenzen, können wir festlegen, dass sich die gemessene Temperatur nur in einer Schritten verändert. Diese Annahmen sollten immer durch Domänenwissen fundiert sein. Dies könnte beispielsweise gegeben sein wenn ein Regler die gemessene Temperatur glättet. Außerdem ist in diesem Szenario die Annahme getroffen worden, dass die Temperatur steigt, wenn der Heizer im Ofen an ist. Ist der Heizer aus, dann fällt die Temperatur.

Das Assumption-Szenario *DeltaTemperature* aus Listing 2.10 schränkt das Verhalten des Temperatursensors, also das Verhalten der Umwelt, ein. Das Szenario wird durch das Senden einer gemessenen Temperatur vom Temperatursensor zum Controller aktiviert. Die gemessene Temperatur wird in einer Variablen zwischengespeichert. Abhängig davon wie sich das System entscheidet (ob der Heizer an oder ausgestellt wird), bedingt sich daraus die nächste gemessene Temperatur des Sensors. Mit der letzten Nachricht aktiviert sich das Szenario wieder selbst.

```
1  assumption scenario DeltaTemperature {
2    var EInt temperature
3    message ts -> ctr.reportTemperature(bind to temperature)
4    alternative {
5      message strict ctr -> heater.turnOn()
6      message strict requested ts -> ctr.reportTemperature(temperature + 1)
7    }or{
8      message strict ctr -> heater.turnOff()
9      message strict requested ts -> ctr.reportTemperature(temperature - 1)
10   }
11 }
```

Listing 2.10: Szenario mit Annahmen über das Verhalten der Umwelt.

2.3.14 Parameterwertebereiche

Für alle Nachrichten können Wertebereiche angegeben werden. Diese gelten für alle Nachrichten desselben Typs. Die Wertebereiche gelten für alle Objekte, die dies Nachricht empfangen. Sie werden auf Ebene der Spezifikation definiert. Alle Nachrichten, die nicht in den Parameterwertebereichen sind können zur Laufzeit nicht gesendet werden.

Listing 2.11 zeigt die Parameterwertebereiche für das Ofen Beispiel. Es werden für drei Nachrichten Parameterwertebereiche definiert. Für alle diese Nachrichten sind die Objekte vom Typ *Controller* als Empfänger zu betrachten. Die erste Nachricht hat einen Parameterwertebereich von 0 bis einschließlich 220.

```
1  parameter ranges {
2    Controller.reportTemperature(tmp = [ 0 .. 220]),
3    Controller.reportNominalTemperature(nominalTemp = [ 0 .. 220 ]),
4    Controller.reportHumidity(hunidity = [ 0 .. 100 ])
5  }
```

Listing 2.11: Parameterwertebereiche für verschiedene Nachrichten

In der Ausführung repräsentiert eine Umweltnachricht mit einem Wildcard-Parameter, zu der eine Parameterbereich definiert wurde, alle konkreten Nachrichten die in diesem Bereich liegen. Listing 2.12 zeigt eine Umweltnachricht mit Wildcard-Parameter. Unter Berücksichtigung der Parameterbereiche aus Listing 2.11 repräsentiert die Nachricht in Zeile 1 alle Nachrichten, die darunter in der Alternative dargestellt sind.

```
1  ts -> ctr.reportTemperature(*)
2  // ist äquivalent zu
3  alternative
4    { ts -> ctr.reportTemperature(0)}
5  or{ ts -> ctr.reportTemperature(1)}
6  // Nachrichten mit den Parametern 2 bis 218
7  or{ ts -> ctr.reportTemperature(219)}
8  or{ ts -> ctr.reportTemperature(220)}
```

Listing 2.12: Konkrete Nachrichten die durch einen Wildcard-Parameter repräsentiert wurden

2.3.15 Verletzungen

Eine Verletzung der Spezifikation kann immer dann auftreten, wenn eine ausgeführte Nachricht nicht mit der aktivierten Nachricht eines Szenarios unifiziert werden kann. Kann anschließend jedoch eine lauschende Nachricht mit einer gesendeten Nachricht unifiziert werden, tritt eine Verletzung auf. Zudem werden die Szenario-Constraints zur Beurteilung hinzugezogen. Sie können Verletzungen aufheben oder sie erzwingen. Für eine genauere Erklärung von Szenario-Constraints siehe 2.3.17.

Ist die aktivierte Nachricht des Szenarios *strict*, führt dies zu einer Sicherheitsverletzung und zum Abbruch der Ausführung an dieser Stellt. Ist die aktivierte Nachricht non-strict, führt dies zu einer *Kalten Verletzung* und das betroffene Szenario wird beendet. Wenn eine Sicherheitsverletzung in einem *Spezifikation Szenario* auftritt, wurde die Spezifikation verletzt und enthält somit einen Fehler. Tritt eine Sicherheitsverletzung in einem Assumption-Szenario auf, dann hat die Umwelt gegen die Annahmen verstoßen. In diesem Fall wird die Ausführung an diesem Zustand nicht fortgesetzt, da die Umwelt den Annahmen zur Folge unrealistisch gehandelt hat. In einer echten Umwelt würde ein solches Verhalten nicht auftreten.

Außerdem kann es sein, dass in einem Systemschritt Nachrichten gefordert sind, die alle gleichzeitig geblockt sind, sodass keine Systemnachricht gesendet werden kann. Dieser Fall kann durch widersprüchliche Anforderungen entstehen und führt ebenfalls zu einer Sicherheitsverletzung.

Eine Nachricht wird von einem Szenario geblockt, wenn dessen Ausführung eine Sicherheitsverletzung bedeuten würde. Systemnachrichten können nur von Spezifikations Szenarios geblockt werden. Genauso können Umweltnachrichten nur von Assumption-Szenarios geblockt werden. Auf diese Weise geblockte Nachrichten können nicht ausgeführt werden.

2.3.16 Sammeln von ausführbaren Nachrichten

Jeder Ausführungsschritt der SML-Spezifikation beginnt damit ausführbare Nachrichten zu identifizieren. Das Aktivitätsdiagramm in Abb. 3.2 erweitert das Aktivitätsdiagramm aus Abb. 2.3 in der Aktivität *Nachrichten sammeln*. In Schritt 1 werden alle von aktiven System Szenarien geforderten Nachrichten gesammelt, also alle Nachrichten aus Spezifikations Szenarien, die aktiviert und gefordert sind.

Ist die Menge nicht leer, dann handelt es sich um einen *Systemschritt*. In diesem Fall werden alle vom System geblockten Nachrichten in Aktivität 2 entfernt. Die Menge der nicht entfernten Nachrichten bildet die Menge der Nachrichten, die im nächsten Schritt ausgeführt werden können. Ist diese Menge leer, so handelt es sich um eine Verletzung der Spezifikation.

Ist die Menge aus Schritt 1 leer, so handelt es sich um einen *Umweltschritt*. In diesem Fall werden alle aktiven geforderten Umweltnachrichten

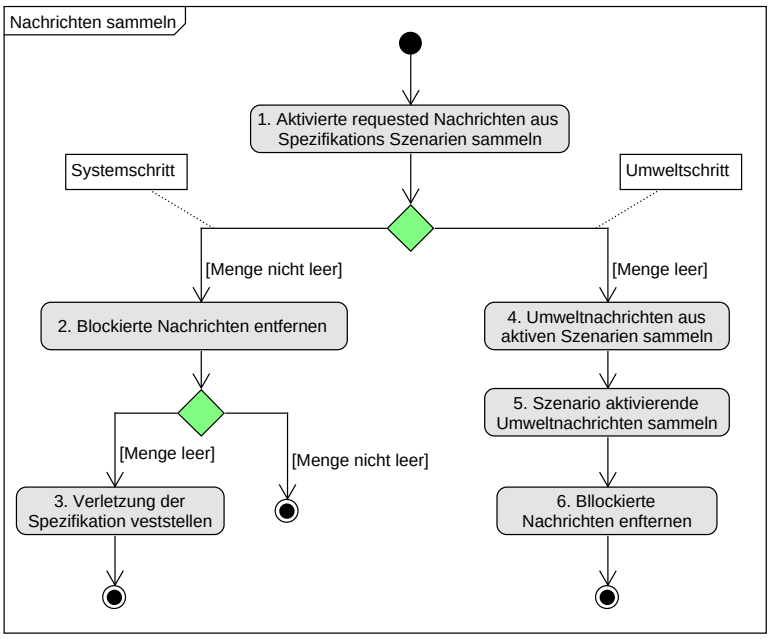

Abbildung 2.4: Sammeln von ausführbaren Nachrichten

aus aktiven Assumption-Szenarios gesammelt. Zusätzlich dazu werden alle Umweltnachrichten gesammelt, die in aktiven Szenarien lauschend sind.

In Schritt 5 werden zudem alle Umweltnachrichten hinzugefügt die Szenarien aktivieren können. Im letzten Schritt werden alle Nachrichten aus der bisherigen Menge aussortiert, die von Assumption-Szenarios geblockt werden oder nicht in den Parameterwertebereichen sind. Zudem wird geprüft, ob Nachrichten mit Wildcard-Parametern in der Menge sind. Existiert für eine dieser Nachrichten ein Parameterwertebereich, so wird die Nachricht durch die Menge der Nachrichten ersetzt, die durch den Parameterwertebereich beschrieben wird. Ansonsten wird diese Nachricht ignoriert.

2.3.17 Szenario-Constraints

Szenario-Constraints sind Einschränkungen oder auch Lockerungen für aktive Szenarien. Durch sie können Szenarien beendet werden oder Sicherheitsverletzungen geworfen werden. Die Syntax von Szenario-Constraints ist in Listing 2.13 dargestellt.

consider: Dieses Constraint fügt eine lauschende Nachricht zum Szenario hinzu. Kann die ausgeführte Nachricht nicht mit der aktivierten Nachricht

aber mit der Constraint Nachricht unifiziert werden, so wird je nach Status der aktiven Nachricht (strict oder non-strict) eine Sicherheitsverletzung geworfen oder das Szenario beendet.

ignore: Dieses Constraint entfernt eine lauschende Nachricht aus dem Szenario. Demnach reagiert das Szenario nur auf diese Nachricht wenn sie aktiviert ist.

interrupt: Kann die ausgeführte Nachricht nicht mit der aktiven Nachricht jedoch mit der Constraint Nachricht unifiziert werden, so wird das Szenario beendet.

forbidden: Kann die ausgeführte Nachricht nicht mit der aktiven Nachricht jedoch mit der Constraint Nachricht unifiziert werden, so wird eine Sicherheitsverletzung geworfen.

```
1 specification scenario Constraints {
2   ...
3 } constraints [
4   consider message sender -> empfaenger.nachricht()
5   ignore message sender -> empfaenger.nachricht()
6   interrupt message sender -> empfaenger.nachricht()
7   forbidden message sender -> empfaenger.nachricht()
8 ]
```

Listing 2.13: Szenario mit Constraints

2.3.18 While und wait-until Konstrukt

Mit dem Sprachkonstrukt *while* wird eine Schleife beschrieben. Dabei wird der Rumpf der Schleife so lange wiederholt bis die Bedingung im Schleifenkopf nicht mehr erfüllt ist. In Listing 2.14 ist eine Schleife zu sehen die zehn Mal ausgeführt wird. Dabei wird die Nachricht *wait()* zehn Mal ausgeführt und anschließend der Zähler um eins erhöht.

```
1 var EInt counter = 0
2 while [counter < 10] {
3   message strict requested ctr -> ctr.wait()
4   counter = counter + 1
5 }
```

Listing 2.14: Ausschnitt der eine Schleife zeigt

Das *wait-until* Konstrukt hält an dieser Stelle die Ausführung des Szenarios solange auf bis diese erfüllt ist. Die Bedingung wird dabei in jedem Ausführungsschritt überprüft. Die Anweisung kann als *strict* markiert werden um anzuzeigen, dass das Szenario beim Warten nicht beendet werden darf. In Listing 2.15 wartet die Ausführung des Szenarios bis das *counter*-Attribut des Controllers größer als 8 ist.

```
1 ...
2 wait until [ctr.counter > 8]
3 ...
```

Listing 2.15: Ausschnitt der eine Wartebedingung zeigt

2.4 ScenarioTools

ScenarioTools[2] ist eine auf Eclipse[3] basierende Entwicklungsumgebung für SML-Spezifikationen. Es baut auf dem Eclipse Modeling Framework (EMF)[4] auf und benutzt Xtext[5] für die Erstellung und Bearbeitung von SML. In ScenarioTools wird der Benutzer bei der Erstellung von Spezifikationen unterstützt. Dabei können Spezifikationen im *Play-Out-Modus* ausgeführt werden. Während der Ausführung werden die Ausführungszustände der einzelnen Szenarien durch das grün hinterlegen der Zeilen markiert. Der erkundete Zustandsraum wird als Graph dargestellt.

Mit der Zustandsraumerkundung kann der Zustandsraum einer Spezifikation automatisch erstellt werden. Der Synthese Algorithmus kann aus dem Zustandsraum ein Teilgraph extrahieren, der als deterministischer Controller für das System dienen kann. Dieser kann im *Stategraph View* betrachtet werden.

In Abbildung 2.5 sind verschiedene Ansichten des Play-Out-Modus zu sehen. Dabei ist auf der rechten Seite der erkundete Zustandsgraph zu erkennen. Wobei der blaue Zustand den aktuellen Zustand darstellt. In der Ansicht unten links werden die in dem aktuellen Zustand erlaubten Nachrichten angezeigt. Links in der Mitte ist die ausgeführte Spezifikation zu sehen, in der der aktuelle Zustand der Szenarien in grün eingefärbt wird. Die Ansicht oben links zeigt die Objekte des Objektsystems, sowie die aktivierten Szenarien des Zustands. Oben in der Mitte ist eine Ansicht zu sehen die die Objektattribute des ausgewählten Objekt anzeigt. In diesem Fall werden die Attribute des Systemobjekts *controller* angezeigt, die in der Ansicht oben links markiert wurden.

In Abbildung 2.6 ist der *StategraphView* zu sehen. Von dieser Ansicht können erkundete Zustandsgraphen, die beispielsweise von einer Zustandsraumerkundung erstellt wurden, angezeigt werden. Der angezeigte Zustandsgraph enthält keine Sicherheitsverletzung, aus diesem Grund sind alle Pfade grün eingefärbt.

[2] http://scenariotools.org/
[3] https://eclipse.org/
[4] https://eclipse.org/modeling/emf
[5] https://eclipse.org/Xtext/

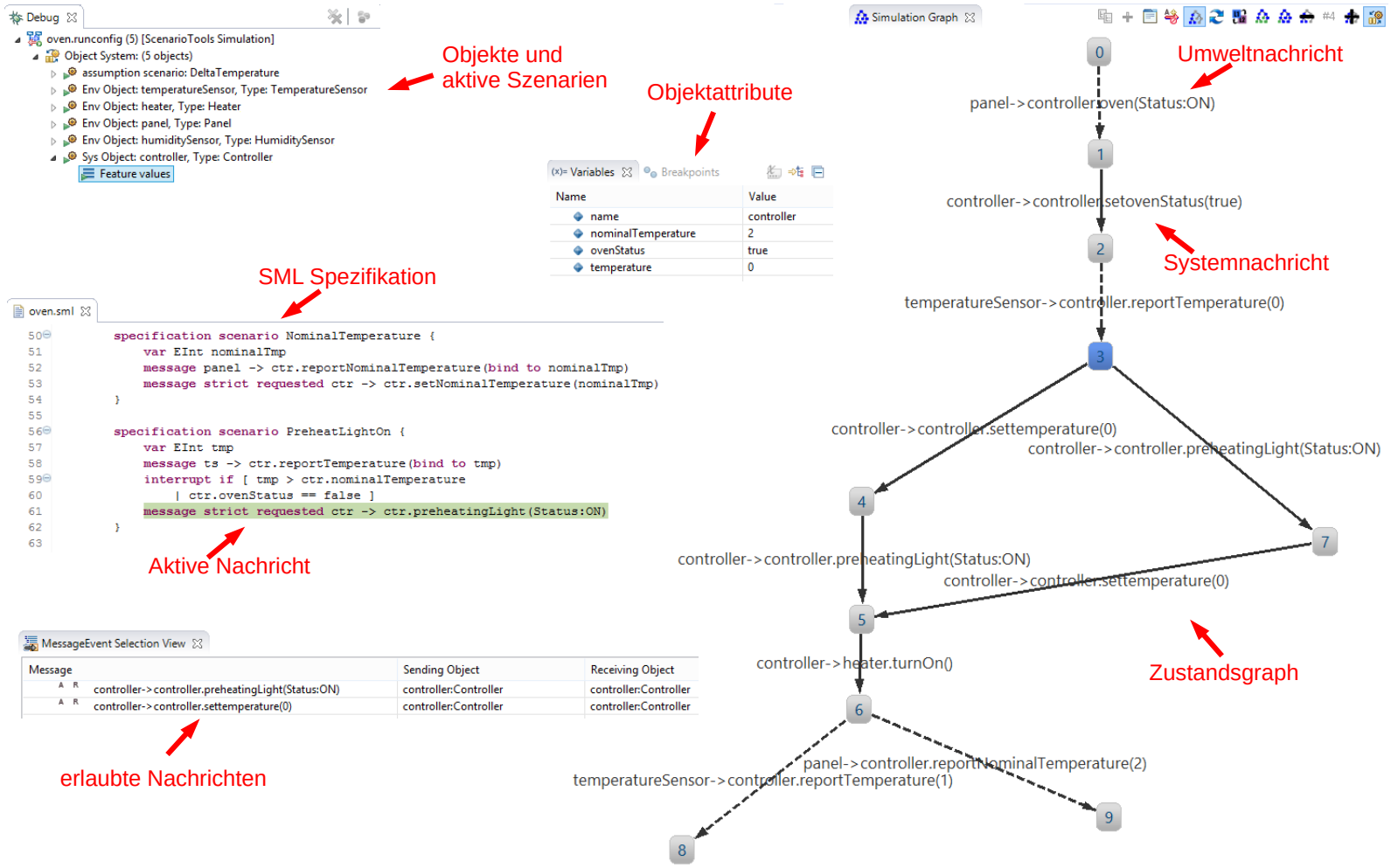

Abbildung 2.5: *Play-Out-Modus* in ScenarioTools

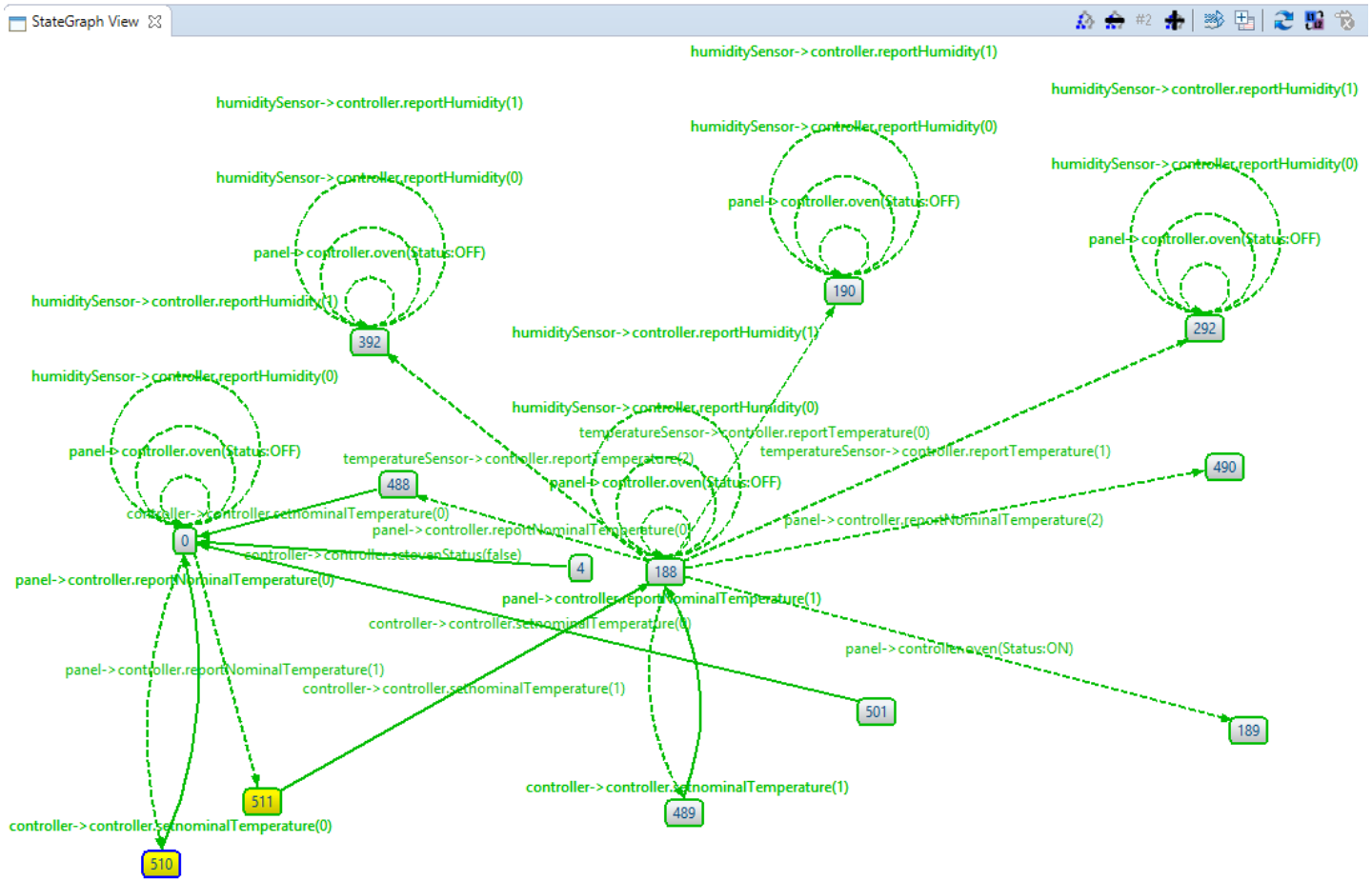

Abbildung 2.6: Screenshot des Stategraph Views

2.5 Zustandsvereinigung bei symbolischer Ausführung

Das Ergebnis der symbolischen Ausführung eines Programmes ist ein symbolischer Ausführungsbaum. Für einfache Programme ohne komplexe Kontrollstrukturen ist dieser Baum endlich. Betrachten wir das Ergebnis der expliziten Zustandsraumerkundung von SML-Spezifikationen, dann stellen wir fest, dass es ein Graph ist. Bei fehlerfreien Spezifikationen hat jeder Zustand einen Folgezustand. Daraus können wir schlussfolgern, dass dieser Graph unendlich lange Pfade enthält. Für die symbolische Ausführung von SML-Spezifikationen bedeutet dies, dass der symbolische Ausführungsbaum unendlich groß wird. Abgesehen davon, dass der Baum unendlich groß wird, werden viele Zustände auch immer und immer wieder erkundet. Im Fall der symbolischen Ausführung bedeuten mehrfach erkundete Zustände keinen neuen Erkenntnisgewinn.

Um diesem Problem entgegen zu wirken können symbolische Zustände vereinigt werden. Es gibt eine Anzahl an verschiedenen Methoden, dies zu tun. Visser et al. [48] haben einige dieser Methoden verglichen und bewertet. Die Methoden können grob in zwei Kategorien unterteilt werden. Es gibt vollständige Verfahren und verlustbehaftete Verfahren.

Zu den vollständigen Verfahren zählt das exakte Zustandsverschmelzen. Dabei werden nur Zustände miteinander verschmolzen, die exakt dieselbe Menge an expliziten Zuständen repräsentieren. Dieses Verfahren gilt als vollständig, da alle erreichbaren Zustände bei der Exploration erkundet werden. Der Zustandsgraph bleibt dabei ebenso vollständig. Er enthält alle Pfade die auch im expliziten Zustandsgraphen vorhanden sind.

Eine weitere Möglichkeit Zustände zu verschmelzen bildet die Durchführung von Teilmengen Tests. Dabei können symbolische Zustände mit anderen symbolischen Zuständen verschmolzen werden, wenn die repräsentierten expliziten Zustände eine Teilmenge des anderen Zustandes darstellen. Auch bei diesem Verfahren werden alle erreichbaren Zustände erkundet. Der wesentliche Unterschied zum vorherigem Verfahren ist im Zustandsgraphen zu erkennen. Durch die Verschmelzung von unterschiedlichen symbolischen Zuständen entstehen *unechte* Pfade im Graphen. Unechte Pfade sind Pfade zu denen kein expliziter Pfad existiert. Das Verfahren im Bezug auf den Zustandsgraphen wird auch als *Überapproximation* bezeichnet. Der Vorteil dieser Methode ist, dass keine Zustände doppelt erkundet werden und der Zustandsgraph dadurch kleiner ausfällt.

Im Gegensatz dazu stehen die verlustbehafteten Verfahren. Dabei wird die Beschreibung der Zustände abstrahiert, um einen Vergleich effizienter zu gestalten. Diese Verfahren werden auch als *Unterapproximation* bezeichnet. Durch Vereinfachungen des Vergleichs, können beim Verschmelzen von symbolischen Zuständen explizite Zustände verloren gehen. Der resultierende

Zustandsgraph kann zum Falsifizieren der Spezifikation verwendet werden. Der Vorteil dieser Verfahren zeichnet sich durch eine gute Performance aus. Die Effizienz dieses Verfahrens hängt sehr stark von der Güte der gewählten Abstraktion ab. Ein Beispiel einer Abstraktion findet sich bei Anand et al. [1].

Alle diese Verfahren haben ihre Stärken und Schwächen. Die Vor- und Nachteile müssen für jede Anwendung separat abgewogen werden. In Kapitel 4 werden einige dieser Verfahren aufgegriffen und diese bei der Anwendung genauer erläutern.

Kapitel 3

Anforderungsanalyse

3.1 Problemanalyse

Betrachten wir das Beispiel der Ofensteuerung aus Abschnitt 2.3: Hier gibt es drei Umweltnachrichten mit Integer-Parametern. Diese besitzen jeweils einen Parameterwertebereich. Wird diese Spezifikation im *Play-Out-Modus* gestartet, dann sind im ersten Zustand 545 ausgehende Transitionen zu erkennen. Diese Menge an ausgehenden Transitionen setzt sich aus den konkreten Nachrichten, welche die Nachrichten durch die Parameterwertebereiche repräsentieren, und den zwei Nachrichten *Ofen an* bzw. *aus*zusammen. Der Zustandsgraph der ersten drei Zustände ist in Abbildung 3.2 dargestellt. Die grauen Knoten symbolisieren die bereits erkundeten Knoten, der blaue Knoten ist der aktuell erkundete Knoten, und die grünen Knoten sind unerkundete Nachfolgeknoten. Dieser Zustandsgraph ist bereits im Anfangszustand nicht mehr zu überblicken. Eine manuelle Erkundung des Zustandsraums erweist sich als unübersichtlich, zeitaufwändig und sehr schwierig. Daraus können wir schließen, dass SML-Spezifikationen, die Umweltnachrichten mit großen Integer-Parameterbereichen besitzen, sehr stark unter der Zustandsraumexplosion leiden.

Eine andere Möglichkeit Rückschlüsse über den Zustandsraum zu erhalten, ist eine automatische Zustandsraumerkundung. Diese Technik leidet jedoch genauso wie die manuelle Erkundung unter dem riesigen Zustandsraum.

Um das Phänomen der Zustandsraumexplosion zu verdeutlichen, schauen wir uns das Beispiel mit reduziertem Parameterwertebereich an (siehe Listing 3.1). Anschließend erhöhen wir den Parameterwertebereich der Nachricht *reportTemperarure(...)* in kleinen Schritten. Die Ergebnisse der Messung sind in Tabelle 3.1 dargestellt. Dabei ist zu erkennen, dass sich der Parameterwertebereich von Messung zu Messung nur leicht vergrößert, die Anzahl der Zustände verdoppelt sich hingegen fast.

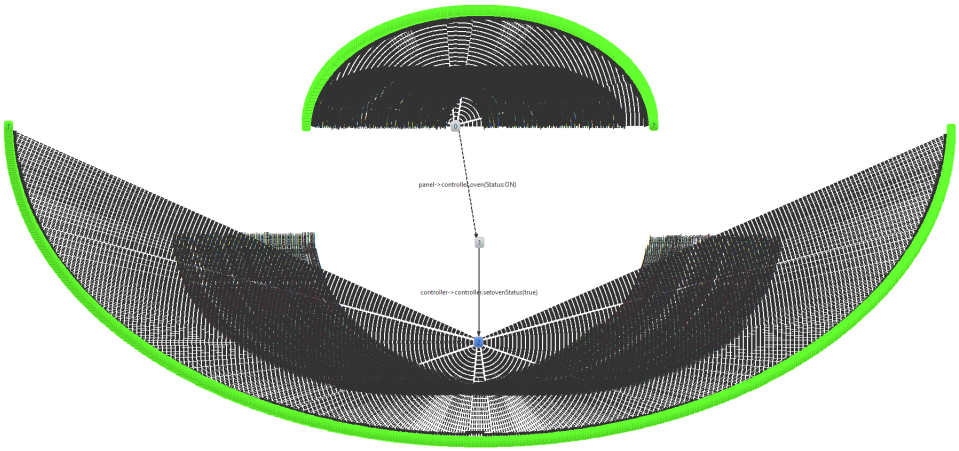

Abbildung 3.1: Alle Nachfolgezustände (grün) der ersten drei Zustände des Ofen-Beispiels, wenn die gesamten Parameterwertebereiche der Anforderung 2.3.2 exploriert werden

Parameterbereich	**#Zustände**	**#Transitionen**	**Zeit (s)**
6 (97 - 102)	339	2661	33
8 (95 - 102)	529	5203	58
11 (95 - 105)	889	11401	128
16 (90 - 105)	1689	30091	347

Tabelle 3.1: Ergebnisse der expliziten Zustandsraumerkundung des Ofen-Beispiels bei unterschiedlich eingeschränkten Parameterbereichen des Temperatursensors

```
1  parameter ranges {
2    Controller.reportTemperature(tmp = [ 97 .. 102]),
3    Controller.reportNominalTemperature(nominalTmp = [ 100 ]),
4    Controller.reportHumidity(humidity = [ 20 .. 22 ])
5  }
```

Listing 3.1: Reduzierte Parameterwertebereiche der Ofensteuerung

Das hieraus resultierende Problem zeichnet sich dadurch aus, dass solche großen Zustandsräume nicht effizient erkundet werden können. Bei diesem Beispiel besteht ein direkter Zusammenhang zwischen der Größe des Parameterwertebereichs und der des Zustandsraumes. Die Komplexität der Spezifikation sollte sich jedoch nicht durch die Größe der Parameterwertebereichen beeinflussen lassen, vielmehr sollte sie sich durch die Logik und Regeln auszeichnen.

3.2 Lösungsansatz

Um dem Problem der Zustandsraumexplosion entgegenzutreten, muss ein Weg gefunden werden, wie die Abhängigkeit zwischen Parameterwertebereichen und dem daraus resultierenden Zustandsraum verringert werden kann, so dass der gesamte Zustandsraum möglichst klein wird.

Eine Möglichkeit dies zu erreichen, ist die Modellierung von Annahmen an die Umwelt in Verbindung mit einer Einschränkung von bestimmten Parameterwertebereichen. Diese Art der Reduzierung der Komplexität wurde bereits im Ofen-Beispiel gezeigt. Es hat sich bei diesem Beispiel jedoch herausgestellt, dass diese Art der Einschränkung nicht ausreichend ist. Ein weiteres Problem bei der Modellierung von Annahmen ist zudem, dass der Benutzer nicht immer realistische Annahmen aufstellt. Ist die Annahme an die Umwelt zu ideal, kann es sein, dass Fehler verborgen bleiben, die in realen Systemen auftreten würden.

In der Problemanalyse haben wir festgestellt, dass eine ungewollte Abhängigkeit zwischen Parameterwertebereichen und dem Zustandsraum besteht. Um diese Abhängigkeit zu lösen, können die Wertebereiche abstrahiert betrachtet werden. Diese Möglichkeit lässt sich mit Hilfe von symbolischer Ausführung umsetzen. Die Stärke von symbolischer Ausführung liegt in der Abstraktion von numerischen Werten.

Betrachten wir das Ofen-Beispiel 2.4: Immer wenn die Nachricht *reportTemperature(*)* auftritt, wird im nächsten Schritt die Temperatur im Objektsystem gespeichert. Das führt dazu, dass jede gespeicherte Temperatur einen neuen Zustand erzeugt. Es ist jedoch nur von Bedeutung, ob die Temperatur kleiner oder größer als die Solltemperatur ist. Daraus lässt sich ableiten, dass nicht die Temperatur an sich für das Verhalten des Systems wichtig ist, sondern vielmehr das Verhältnis zwischen Temperatur und Solltemperatur. Aus diesem Grund bietet es sich an, diese konkreten Werte durch Abstraktion zu eliminieren.

Nun gibt es bereits eine große Fülle an symbolischen Ausführungsprogrammen. Demnach sollte zuerst geprüft werden, ob ein bereits existierendes Tool adaptiert werden kann. (Für eine ausführliche Auflistung von ähnlichen Programmen siehe Kapitel 7.) Es gibt bereits einige Arbeiten, in denen Modelle in Eingabesprachen von symbolischen Ausführungsprogrammen transformiert werden. Ein Beispiel ist das Programm *Symbolic PathFinder*(SPF), das automatisch eine Transformation von *Simulink/Stateflow*, *Standard UML* oder *Rhapsody UML* zu JAVA mit Hilfe des *Polyglot Tools* durchführen kann. Anschließend können diese Modelle symbolisch analysiert werden [41, 42].

Im Prinzip ist SML eine erweiterte Form von UML Sequenzdiagrammen. Der Unterschied in der Interpretation zwischen den beiden Modellierungssprachen ist jedoch groß. Die Implementierung einer Transformation für SML impliziert die Neuimplementierung der Ausführungslogik in einem Trans-

formationsalgorithmus. Dies ist einer Neuimplementierung von ScenarioTools gleichzusetzen. Zudem erhöht dies den Wartungsaufwand drastisch, da nun Änderungen in der Semantik oder Syntax in zwei Programmen angepasst werden müssten.

Eine weitere Möglichkeit wäre es die gesamte Synthese oder Zustandsraumerkundung von einem Tool wie SPF symbolisch testen zu lassen. Bei dieser Vorgehensweise ist die Gefahr einer Zustandsraumexplosion genauso gegeben wie bei der konkreten Ausführungslogik. Dies begründet sich dadurch, dass nun nicht die Spezifikation an sich analysiert wird, sondern der Teil von ScenarioTools der die Spezifikation ausführt.

Die dritte Möglichkeit ist die Integration der symbolischen Ausführung in die Ausführungslogik unserer eigenen Entwicklungsumgebung ScenarioTools. Dies bietet die Möglichkeit die symbolische Ausführung direkt auf unsere Bedürfnisse anzupassen. Zudem bleibt ScenarioTools auf diese Weise wartbar. Es kann sicher gestellt werden, dass bei einer Zustandsraumerkundung nur die Spezifikation symbolisch getestet wird und nicht die Ausführungslogik. Zudem kann auf einfache Weise definiert werden, welcher Teil der Spezifikation symbolisch analysiert werden soll. Der Arbeitsaufwand einer eigenen Implementierung ist sehr viel geringer als der für eine Transformation. Dann steht natürlich nur eine Teilmenge der Funktionalität zur Verfügung stehen, die zum Beispiel SPF bieten könnte. Dem gegenüber steht, die Freiheit die symbolische Ausführung direkt an die Bedürfnisse von szenariobasierten Spezifikationen anpassen zu können.

3.3 Vision

Ziel der symbolischen Ausführung ist es, einen Zustand so abstrakt darzustellen, dass Zustände aus denen gleiches Verhalten resultiert zusammengefasst werden. Betrachten wir noch einmal die Ofensteuerung. Um das Beispiel übersichtlich zu halten, reduzieren wir das Szenario *OvenRegulation* 2.4 auf die Entscheidung, ob die gemessene Temperatur wärmer oder kälter als eine feste Solltemperatur ist. In der konkreten Ausführung ist der Ergebniszustandsraum wie ein Stern aufgebaut. Zu jedem Wert, der gemessen und anschließend getestet wird, geht aus der Startzustand eine Transition zu einem neuen Zustand und von diesem geht eine Transition zurück zum Startzustand. Ansonsten macht diese Spezifikation nichts anderes. Der resultierende Zustandsraum ist auf der rechten Seite der Abbildung 3.2 dargestellt. Symbolisch betrachtet werden nun alle Zustände, in denen die gemessene Temperatur wärmer als die Solltemperatur ist, zu einem Zustand zusammengefasst. Zudem werden alle Transitionen zusammengefasst, die zu einem Zustand führen, bei dem die übermittelte Temperatur wärmer als die Solltemperatur ist. Genauso sollen alle Transitionen und Zustände zusammengefasst werden, bei denen die Temperatur niedriger als die Soll-

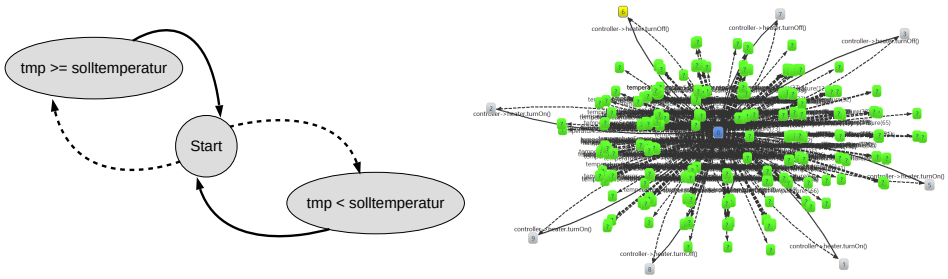

Abbildung 3.2: Zustandsraum des Szenarios *OvenRegulation*. Links: Reduzierung auf die wesentlichen Zustände. Rechts: Konkreter Zustandsraum

temperatur ist. Wie das Ergebnis aussehen könnte ist in der linke Abbildung in 3.2 dargestellt.

Die Abstraktion ermöglicht nun eine zielführende Analyse von SML-Spezifikationen mit großen Parameterwertebereichen möglich. Dabei beschreiben ausgehende Transitionen eines Zustandes unterschiedliches Verhalten. Dadurch ist die Simulation im *Play-Out-Modus* auf das wesentliche reduziert und dementsprechend übersichtlicher. In der automatisierten Zustandsraumerkundung werden insgesamt wesentlich weniger Zustände erkundet. Dadurch können komplexe Spezifikationen effizient exploriert werden, die bei der expliziten Ausführung praktisch nicht explorierbar waren. Die Ergebnisse der Zustandsraumerkundung können (wie bei der expliziten Variante) als Zustandsgraph betrachtet werden, in dem die wichtigen Informationen dargestellt werden. Zudem können wichtige Pfade extrahiert werden. Ein Beispiel sind Pfade zu Sicherheitsverletzungen, die bei der symbolischen Ausführung erkundet wurden. Aus diesen Pfaden können anschließend konkrete Pfade synthetisiert werden. Solche Pfade können als Gegenbeispiel im expliziten *Play-Out-Modus* ausgespielt werden und an konkreten Zuständen nachvollzogen werden.

3.4 Anforderungen

Die Ausführungslogik von ScenarioTools soll um eine symbolische Ausführung erweitert werden. Daraus ergeben sich eine Reihe von Anforderungen. In diesem Abschnitt wird erhoben, welche Anforderungen an eine Erweiterung gesellt werden müssen, damit eine symbolische Ausführung von SML-Spezifikationen möglich wird. Dazu muss zum einen geklärt werden, in welchem Umfang eine symbolische Ausführung unterstützt werden kann und zum anderen wie der Benutzer diese Bereiche kennzeichnen kann. Das Ziel der symbolischen Ausführung soll die Akkumulation von Zuständen sein, die gleiches Verhalten fordern und sich lediglich in der Wertebelegung von

Parametern, Szenariovariablen oder Objektattributen unterscheiden.

1. **Anforderung** Für die Definition symbolischer Parameter soll die Konfigurationsdatei so erweitert werden, dass einzelne Parameter bestimmter Nachrichten als symbolisch definiert werden können.

2. **Anforderung** Für die Definition symbolische Objektattribute soll die Konfigurationsdatei so erweitert werden, dass einzelne Attribute bestimmter Objekttypen als symbolisch definiert werden können.

3. **Anforderung** Diese symbolischen Parameter müssen von der SCENARIO-TOOLS Ausführungslogik verarbeitet werden können. Wir beschränken uns auf den Typ *Integer* als erlaubten symbolischen Parameter.

4. **Anforderung** Die Simulation von symbolischen Spezifikationen soll möglich sein.

5. **Anforderung** Die Erkundung des gesamten Zustandsraums soll automatisch möglich sein.

6. **Anforderung** Szenarien müssen symbolische Parameter verarbeiten können. Das schließt das Speichern von symbolischen Parametern in Szenariovariablen und das Auswerten von Bedingungen mit symbolischen Ausdrücken mit ein.

7. **Anforderung** Das Objektsystem soll während der Ausführung symbolische Werte vom Typ *Integer* in Objektattributen speichern können.

8. **Anforderung** Zustände die gleich sind oder eine Teilmenge eines anderen Zustandes abbilden sollen als solche erkannt und als ein abstrakter Zustand dargestellt werden.

9. **Anforderung** Wenn es während der symbolischen Ausführung einer Nachricht zu einer Aufsplittung des Nachfolgezustands kommt, soll die verschiedenen Möglichkeiten automatisch erkundet werden.

Ziel dieser Arbeit soll die Abstraktion von numerischen Eingaben aus der Umwelt sein. Diese Eingaben werden durch parametrisierte Umweltnachrichten repräsentiert. Auf diese Weise können Gruppen von Zuständen zusammengefasst werden. Darüber hinaus kann auf diese Weise die teils aufwendige Modellierung der Umwelt reduziert werden.

Kapitel 4

Konzept

Die Grundidee des Konzepts für eine symbolische Ausführung in ScenarioTools ist stark an die im Grundlagenkapitel 2.1 beschriebene symbolische Ausführung von Programmen angelehnt. Im Fall von ScenarioTools gibt es ebenfalls Eingaben von außen, die mit symbolische Parametern dargestellt werden, sowie *Pfad-Constraints*. Diese werden hier im Zusammenhang mit ScenarioTools *Zustands-Constraints* genannt.

Es gibt jedoch auch Unterschiede zur klassischen symbolischen Ausführung. Ein Unterschied ist die Tatsache, dass die Szenarien nicht wie normale Funktionen sequenziell abgearbeitet werden, sondern eine Nachricht mehrere Szenarien betrifft und diese parallel ausgeführt werden. Damit sind Bedingungen, die in einem Szenario gültig werden, mit Seiteneffekten für andere Szenarien verbunden.

Ein weiterer wesentlicher Unterschied ist die Auslegung der betrachteten Systeme als unendlich laufende reaktive Systeme. Ausgeführte Spezifikationen besitzen daher keinen Endzustand. Der symbolische Ausführungsbaum wird dadurch unendlich groß. Dies verhindert die Terminierung der symbolischen Ausführung von szenariobasierten Systemen. Aus diesem Grund ist es wichtig, eine Form der Zustandsvereinigung zu unterstützen, um auf diese Weise die Ausführung endlich zu machen. Durch das Vereinigen von Zuständen wird der Ausführungsbaum zurück in die gewohnte Graphenform gebracht.

4.1 Symbolischer Interpreter

Bei der Umsetzung eines Konzepts für die symbolische Ausführung ist es existenziell wichtig eine Einheit zu besitzen, die symbolische Bedingungen und Constraints auswerten kann. Diese Aufgabe übernimmt ein symbolischer Interpreter, der alle benötigten Funktionen zur Verfügung stellt. Zu den wichtigsten Funktionen zählen die Auswertung von Bedingungen unter Berücksichtigung der Zustands-Constraints, sowie das Testen von Relationen

zwischen verschiedenen Constraints.

Im Kern des symbolischen Interpreters befindet sich ein Solver. Wobei der symbolische Interpreter die Zustands-Constraints und die Bedingungen aus Szenarien etc. interpretiert und auf mathematische Formeln herunterbricht. Diese werden dann an den Solver weitergereicht.

Der symbolische Interpreter übernimmt damit die Aufgabe des Bindeglieds zwischen Ausführungslogik für SML-Spezifikationen und generischem Solver. Implementierungsdetails sind in Abschnitt 5.3 zu finden.

4.2 Symbolische Nachrichten

Umweltnachrichten, die Parameter vom Typ Integer besitzen, können in der Konfiguration als symbolische Parameter definiert werden (siehe Listing 4.1). Dabei wird eine Nachricht eines Empfängertyps ausgewählt und die gewünschten Integer-Parameter auf symbolisch gesetzt. Auf diese Weise können auch einzelne Parameter einer Nachricht als symbolisch definiert und die anderen weiter als konkrete Integer-Parameter behandelt werden.

```
1 symbolic parameters {
2   Controller.reportTemperature(tmp:symbolic)
3 }
```

Listing 4.1: Definition von symbolischen Nachrichten in der Konfiguration

Gibt es während der Ausführung Umweltnachrichten, die einen Wildcard-Parameter besitzen, der als symbolisch definiert wurde, dann wird diese Nachricht durch eine symbolische Nachricht ersetzt und somit der Wildcard-Parameter durch eine neue symbolische Variable ersetzt. Nachrichten mit dem *bind to* Schlüsselwort gelten auch als Wildcard-Parameter.

Nachrichten die während der Ausführung einen konkreten Parameterwert oder bereits einen symbolischen Wert besitzen, werden nicht durch neue symbolische Variablen ersetzt. Durch ein Szenario kann einer Nachricht ein symbolischer Wert zugewiesen werden ohne, dass dieser Parameter als symbolisch definiert wurde.

4.3 Symbolische Objektattribute

Symbolische Attribute von Objekten aus dem Objektsystem können über *set*-Nachrichten definiert werden. Dazu werden der Typ des Objekts und das Attribut ausgewählt, welches bei der Initialisierung als symbolisch definiert werden soll. Dies ist in Listing 4.2 zu sehen. Dabei kann das Objektsystem auch als eine Form der symbolischen Eingabeparameter betrachtet werden, wobei hier nur einzelne Objektattribute symbolisch betrachtet werden.

```
1 symbolic attributes {
2   Controller.setTemperature(temperature:symbolic)
3 }
```

Listing 4.2: Definition von symbolischen Attributen in der Konfiguration

Während der Ausführung können Objektattribute neue Werte mittels *set*-Nachrichten zugewiesen werden. Diese können konkret oder symbolisch sein, auch wenn sie vorher nicht als symbolisch definiert wurden.

4.4 Symbolische Szenariovariablen

Einer Szenariovariable kann einen symbolischer Wert zugewiesen werden, indem ein symbolischer Parameterwert einer Nachricht mittels eines *bind to*-Befehls gebunden wird (siehe Zeile 3 von Listing 4.3). Eine zweite Möglichkeit besteht darin der Szenariovariablen ein symbolischen Wert eines Objektattributs zu zuweisen (siehe Zeile 4 von Listing 4.3). Diese symbolischen Szenariovariablen können wiederum wie normale Szenariovariablen als Parameter für Nachrichten eingesetzt werden (siehe Zeile 5 von Listing 4.3).

```
1 specification scenario SymbolicVars {
2   var EInt symbolicVar
3   message env -> ctr.reportTemperature(bind to symbolicVar)
4   symbolicVar = ctr.symbolicIntegerVar
5   message ctr -> ctr.setTemperature(symbolicVar)
6 }
```

Listing 4.3: Symbolische Werte in Szenariovariablen

4.5 Zustands-Constraints

Zustands-Constraints beschreiben die Wertebereiche von symbolischen Variablen und ihre Abhängigkeiten untereinander. Während in Szenarien und Nachrichten nur die symbolischen Variablen referenziert werden, so werden im Zustand ihre Bedeutung gespeichert. Durch Austausch der Zustands-Constraints lassen sich die Bedeutungen der Variablen verändern.

```
1 (5 >= tmp & tmp <= 100), !(tmp = 50), (tmp < 88)
```

Listing 4.4: Zustand Constraints, die die symbolische Variable *tmp* beschreiben

Die Zustands-Constraints in Listing 4.4 beschreiben den Wert der symbolischen Variable *tmp*. Sie liegt im Wertebereich von 5 bis einschließlich 87 und ist ungleich 50.

4.6 Symbolische Parameterunifizierung und Splitbedingung

Bei der Unifizierung von Nachrichten mit symbolischen Parametern müssen generell drei Fälle beachtet werden. Diese hängen von den Kombinationen von symbolischen und konkreten Nachrichten ab. Dabei wird zwischen der *gesendeten Nachricht* und der *aktivierten Nachricht* unterschieden. Die gesendete Nachricht stellt die Nachricht dar, die von der Umwelt oder dem System zur Ausführung ausgewählt wurde. Die aktivierte Nachricht stellt eine in einem Szenario aktivierte Nachricht dar. Die Parameterunifizierung mit lauschenden Nachrichten oder Szenario-Constraint Nachrichten eines Szenarios verhält sich identisch.

Während der Parameterunifizierung werden die entsprechenden Parameter zweier Nachrichten auf Gleichheit bzw. Ungleichheit überprüft. Da die Parameter der symbolischen Nachrichten mehrere konkrete Werte repräsentieren können, ist das Ergebnis der Unifizierung nicht immer eindeutig. Sind beide Fälle möglich, dann ist die *Splitbedingung* erfüllt. Eine Splitbedingung ist genau dann erfüllt, wenn die Auswertung einer symbolischen Bedingung unter Berücksichtigung der Zustands-Constraints *wahr* ergibt und die Auswertung der negierte Bedingung unter Berücksichtigung der Zustands-Constraints *wahr* ergibt. Wenn dies erfüllt ist, müssen beide Pfade weiter verfolgt werden: Der Pfad, in dem die Parameterunifizierung erfolgreich war, und der in dem sie es nicht war. In beiden Fällen werden die Zustands-Constraints mit einem Constraint erweitert, das die erfolgreiche bzw. nicht erfolgreiche Unifizierung beschreibt.

Fall 1: *Die gesendete Nachricht ist symbolisch und die aktivierte Nachricht ist symbolisch.*
In diesem Fall kann es sein, dass die beiden Nachrichten die gleiche Menge an konkreten Nachrichten repräsentieren. Wenn die Menge größer als ein Element ist, können die Parameter gleich oder ungleich sein. Das Gleiche gilt für zwei Mengen die eine Schnittmenge besitzen. Bei einelementigen Mengen verhält sich die symbolische Parameterunifizierung wie im konkreten. Sind die Mengen disjunkt, ist das Ergebnis nicht unifizierbar.

Fall 2: *Die gesendete Nachricht ist konkret und die aktivierte Nachricht ist symbolisch.*
Dieser Fall kann als Sonderfall von Fall 1 betrachtet werden bei dem die gesendete Nachricht einelementig ist. Ist der symbolische Parameter der aktivierten Nachricht ebenfalls einelementig, verhält sich die Unifizierung wie im konkreten Fall. Ist der Wertebereich des symbolischen Parameters größer, dann wird bei der Unifizierung auf eine Schnittmenge der beiden Mengen geprüft. Existiert eine Schnittmenge, ist die Splitbedingung erfüllt. Gibt es keine Schnittmenge, ist das Ergebnis nicht unifizierbar.

Fall 3: *Die gesendete Nachricht ist symbolisch und die aktivierte Nachricht ist konkret.*
Dieser Fall verhält sich wie Fall 2. Allerdings muss ein Sonderfall berücksichtigt werden. Wenn die aktivierte Nachricht einen Wildcard-Parameter besitzt, dann ist das Ergebnis immer unifizierbar.

Diese Beschreibung ist für Nachrichten mit einem Parameter ausgelegt. Besitzt die Nachricht mehrere Parameter, dann wird die Ergebnismenge durch die Kombination aller Einzelergebnisse der Parameter konstituiert. Hierbei ist sind zwei Nachrichten nicht unifizierbar, wenn ein Parameter nicht unifizierbar ist. Die Splitbedingung ist erfüllt sobald sie für einen Parameter erfüllt ist und alle weiteren Parameter unifizierbar sind. Siehe dafür auch Abschnitt *Multi-Split* 4.9.

Betrachten wir folgendes Beispiel 4.5: Die Ausführung befindet sich in einem Zustand mit folgenden Zustands-Constraints ZC. Der Parameter der gesendeten Nachricht ist *tmp*. Dieser ist symbolisch und nicht einelementig. Das Gleiche gilt für den Parameter *tmp2* der aktivierten Nachricht. Bei der Parameterunifizierung wird geprüft, ob die Parameter unter der Berücksichtigung der Zustands-Constraints gleich sind: $tmp = tmp2$. Bei Betrachtung der Zustands-Constraints ZC wird festgestellt, dass die Formel erfüllbar ist. Im nächsten Schritt wird geprüft, ob die Parameter unterschiedlich sind: $tmp \neq tmp2$. Auch diese Bedingung ist unter Berücksichtigung der Zustands-Constraints ZC erfüllbar. Damit ist die Splitbedingung erfüllt und es werden zwei Nachfolge Zustände erstellt. Dabei werden die Zustands-Constraints des ersten Folgezustands $ZC1$ um die Bedingung $tmp = tmp2$ erweitert. Die Zustands-Constraints des zweiten Folgezustands $ZC2$ werden um die negierte Bedingung $tmp \neq tmp2$ erweitert.

```
1 Zustands-Constraints: ZC := (5 >= tmp & tmp <= 100) (tmp2 < 50)
2 gesendete Nachricht reportTemperature(tmp)
3 aktivierte Nachricht reportTemperature(tmp2)
4
5 Nachfolge Zustand 1: erfolgreiche unifiziert
6 Zustands-Constraints: ZC1 := (5 >= tmp & tmp <= 100), (tmp2<50), (tmp=tmp2)
7
8 Nachfolge Zustand 2: nicht unifiziert
9 Zustands-Constraints: ZC2 := (5 >= tmp & tmp <= 100), (tmp2<50), !(tmp=tmp2)
```

Listing 4.5: Zustand mit zwei Folgezuständen, die durch eine Splitbedingung entstanden sind

4.7 Szenario-Constraints

Szenario-Constraints werden geprüft, wenn während der Ausführung eines Szenarios die gesendete Nachricht nicht erfolgreich mit der aktivierten Nachricht unifiziert werden konnte. Das Prüfen der Szenario-Constraints erfolgt nach den Regeln der Parameterunifizierung. Wird die *Splitbedingung* (siehe Abschnitt 4.6) erfüllt, dann wird auch hier ein Split durchgeführt.

Das Listing 4.6 zeigt ein Beispiel-Szenario, in dem die zwei Nachrichten in der Alternative aktiviert sind. Zudem besitzt das Szenario eine Constraint-Nachricht. Wird nun die symbolische Nachricht

$$controller \rightarrow controller.reportTemperature(tmp)[tmp > 0 \wedge tmp < 220]$$

ausgeführt, dann werden zuerst die aktivierten Nachrichten auf Unifizierbarkeit geprüft. Dies ist in dem Beispiel nicht erfolgreich. Anschließend werden die Szenario-Constraints geprüft. An dieser Stelle wird die Parameterunifizierung mit der gesendeten und der Constraint-Nachricht durchgeführt. Dabei ist die Splitbedingung erfüllt. Für den Fall der erfolgreichen Unifizierung wird das Constraint $tmp = 1$ zu den Zustands-Constraints hinzugefügt und das aktivierte Szenario beendet. Im Falle der nicht erfolgreichen Unifizierung wird das Constraint $tpm \neq 1$ hinzugefügt und das Szenario bleibt unverändert.

```
1  specification scenario PositiveOrNegative {
2    var EInt counter
3    alternative {
4       message strict controller -> controller.isPositive()
5    } or {
6       message strict controller -> controller.isNegative()
7    }
8  } constraints [
9    interrupt message controller -> controller.reportTemperature(1)
10 ]
```

Listing 4.6: Szenario mit zwei aktivierten Nachrichten und einer Szenario-Constraint Nachricht

4.8 Bedingungen

Bedingungen die während der Ausführung symbolisch geprüft werden, müssen auf die *Splitbedingung* (siehe Abschnitt 4.6) getestet werden. Dabei sind bei folgende Konstrukten die ausgeführten Bedingungen zu beachten:

```
1  interrupt if [Bedingung]
2  violation if [Bedingung]
3  alternative if [Bedingung] { ... } or if [Bedingung] { ... }
4  while [Bedingung] { ... }
5  wait until [Bedingung]
```

Listing 4.7: Konstrukte mit Bedingungen, die einen Split erzeugen können

Die Folgen eines Splits reichen damit von der Beendigung eines Szenarios über das Auftreten einer Sicherheitsverletzung bis hin zur Aktivierung von Nachrichten.

4.9 Multi-Split

Während der Ausführung eines Schrittes kann es zu mehreren Splits kommen.

Split	Ausführung	Pfad 1	Pfad 2
1	$(tmp = tmp2)$	$!(tmp = tmp2)$	–
2	$(tmp = tmp2)$ $(tmp < 30)$	$!(tmp = tmp2)$	$(tmp = tmp2)$ $!(tmp < 30)$

Tabelle 4.1: Constraint-Kombination bei mehreren erfüllten Splitbedingungen pro Ausführungsschritt

Dies ist beispielsweise der Fall, wenn eine Bedingung auf eine Nachricht folgt. Es ist möglich, dass sowohl bei der symbolischen Parameterunifizierung als auch bei der Prüfung einer symbolischen Bedingung direkt im Anschluss die Splitbedingung erfüllt wird. Während der Ausführung werden die Informationen von Splits gesammelt. Die Ausführung folgt bei einer Splitbedingung immer dem nicht negierten Fall. Die zusätzlichen Pfade werden erst nach Beendigung eines Schrittes ausgeführt. Anschließend wird der Schritt mit den zurückgestellten Constraints erneut ausgeführt.

Zur Verdeutlichung dieses Konzeptes folgt ein Beispiel. Zu Beginn eines Ausführungsschrittes gibt es eine Startmenge an Zustands-Constraints:

$$S1 : (5 >= tmp), (tmp <= 100), (tmp2 < 50)$$

Während der Ausführung wird als Erstes eine Parameterunifizierung durchgeführt. Die daraus entstandenen Constraints sind in der Tabelle 4.1 bei *Ausführung*, erster *Split* sowie *Pfad 1* erster *Split* zu sehen. Anschließend wird eine symbolische Bedingung geprüft. Auch hier ist die Splitbedingung erfüllt. Die resultierenden Zustands-Constraints sind in der Zeile des zweiten Splits in den Spalten *Ausführung* sowie *Pfad 2* zu sehen. Damit sind zwei neue Pfade entstanden die noch nicht exploriert wurden. Durch diese Art der Constraint-Erstellung ist es ausgeschlossen, dass Pfade in einem Schritt mehrfach erstellt werden. Die gesamten Zustands-Constraints setzen sich aus der Startmenge der Constraints und aus dem Splitbedingungen zusammen.

Die Constraints beschreiben Pfade durch die Spezifikation, die noch nicht durchlaufen wurden. Es kann sein, dass bei der Erstellung dieser Pfade ungültige Pfade entstehen. Dies ist ein Sonderfall, der beispielsweise bei Nachrichten mit mehreren symbolischen Parametern auftreten kann. Aus diesem Grund werden nach Beendigung des Schrittes alle neu erzeugten Pfad-Constraints auf Erfüllbarkeit geprüft. Alle Pfade, die nicht erfüllbar sind, werden verworfen. Dies ist eine Form der Pfadbeschneidung zur Vermeidung von Erkundungen nicht gültiger Pfade.

4.10 Parameterwertebereiche

Wenn ein Nachrichtentyp als symbolisch gekennzeichnet ist und gleichzeitig Parameterwertebereiche für ihn angegeben sind, dann werden die Parame-

terbereiche als Constraints für diesen Parameter übernommen.

```
1 ts -> ctr.reportTemperature(tmp)[(tmp >= 0 & tmp <= 220)]
```

Listing 4.8: Parameterbereiche werden zu Zustands-Constraints, die die symbolische Variable *tmp* einschränken

Die Nachricht aus Listing 4.8 ist eine Umweltnachricht mit einem Wildcard-Parameter. Während des Sammelns neuer Nachrichten wurde eine neue symbolische Variable erstellt. Da es einen Parameterbereich für diesen Nachrichtentyp gibt, wird dieser ebenfalls als Constraint zu den Zustands-Constraints hinzugefügt.

4.11 Listen Operationen

In ScenarioTools werden eine Reihe von Listen Operationen unterstützt. In dieser Arbeit soll die *Liste.get(index)*-Funktion betrachtet werden. Mit dieser Funktion kann über den Index auf ein Element zugegriffen werden. Wenn der gegebene Index eine symbolische Variable ist, wird für jeden möglichen Index ein Split ausgeführt. Da hier symbolische Werte während der Ausführung zu konkreten Werten umgewandelt werden, handelt es sich hierbei um eine Form der *Concolic Execution* [20, 45, 11].

Listing 4.9 zeigt eine SML-Spezifikation, in der das System eine Liste von Namen verwaltet. Die Liste enthält sechs Einträge. Die Umwelt kann diese Namen mit einer ID erfragen. Das Assumption-Szenario *askForName* erfragt mit der Nachricht *getNameForId(...)* einen Namen mit einer ID aus dem Wertebereich von 0 bis 7. Anschließend wartet das Szenario auf die Beantwortung dieser Anfrage. Das Szenario *answerNames* nimmt die Anfrage der Umwelt entgegen. Im ersten Schritt wird geprüft ob die ID größer gleich dem letzten Element in der Liste ist. Ist dies der Fall, wird die Anfrage nicht weiter beantwortet. Anderenfalls wird der zur ID passende Name aus der Liste geholt und zurück an die Umwelt gesendet. Erfolgt die Anfrage an die Liste mit einem symbolischen Wert, dann wird bei der *get*-Anfrage nach einer konkreten Lösung für den symbolischen Wert gesucht. Gibt es dabei mehr als eine mögliche Lösung, dann verhält sich die Anfrage wie eine erfüllte Splitbedingung. Das heißt es gibt zwei neue Bedingungen für die Zustands-Constraints. Angenommen die erste Lösung für den symbolischen Wert ist 1, dann sind die neu entstandenen Constraints die folgenden: $id = 1$ und $id \neq 1$. Anschließend werden die neu entstehenden Pfade solange erkundet bis es nur noch eine mögliche Lösung für den symbolischen Wert gibt. Dadurch werden alle möglichen Elemente der Liste zurück gegeben. Abbildung 4.1 zeigt den bei der Rückgabe aller Listenelemente entstandenen (Teil-)Zustandsgraphen.

```
1  parameter ranges {
2    Controller.getNameForId(index = [ 0..7])
3  }
4
5  collaboration Names {
6
7    dynamic role Controller ctr
8    dynamic role Environment env
9
10   specification scenario answerNames {
11     var EInt id
12     message env -> ctr.getNameForId(bind to id)
13     interrupt if[ ctr.names.size() <= id]
14     var EString name = ctr.names.get(id)
15     message strict requested ctr -> env.returnName(id, name)
16   }
17
18   assumption scenario askForName {
19     var EInt id
20     message env -> ctr.getNameForId(bind to id)
21     message requested ctr -> env.returnName(id, *)
22   }
23 }
```

Listing 4.9: Zugriff auf eine List mit Strings durch einem symbolischen Index

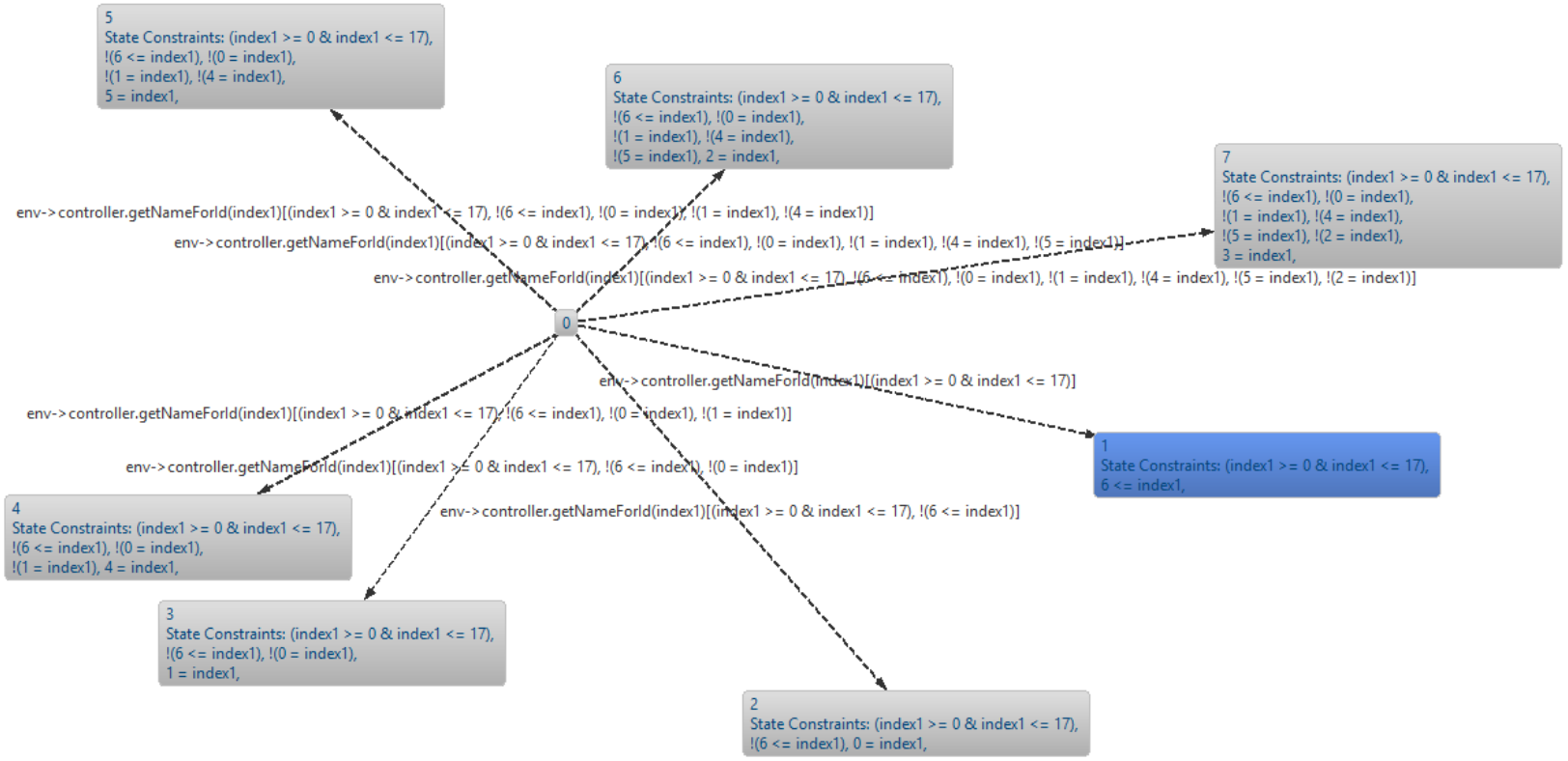

Abbildung 4.1: (Teil-)Zustandsgraphen der bei der Rückgabe aller Listenelemente entsteht

4.12 Symbolischer Zustand

Ein symbolischer Zustand repräsentiert eine Menge konkreter Zustände, die durch die Zustands-Constraints beschrieben wird. Er zeichnet sich dadurch aus, dass er symbolische Werte besitzt. Das heißt, dass mindestens eine Szenariovariable oder ein Objektattribut symbolisch ist. Symbolische Parameter von Nachrichten zählen hier nur indirekt mit hinein. Entweder werden sie in einem Szenario durch eine Szenariovariable bzw. ein Objektattribut bestimmt, oder sie kommen aus der Umwelt und haben erst Bedeutung, wenn sie im folgenden Schritt ausgeführt werden.

Ein symbolischer Zustand S wird durch das Quadrupel $S = (s, Sz, OS, SC)$ beschrieben.

- s stellt den Zustand im konkreten Sinne dar. Der Zustand wird durch den Ausführungszustand, die Rollenbindungen, die Belegung der konkreten Variablen und das konkrete Objektsystem beschreiben. (Vergleiche dazu Abschnitt 2.3.10). Die Werte in den Szenarien und Objekten, die symbolisch sind, werden für den konkreten Vergleich als solche markiert und nicht genauer betrachtet
- Sz ist die Menge der symbolischen Variablen in aktiven Szenarien.
- OS bezeichnet die Menge der symbolischen Variablen im Objektsystem.
- SC steht für die Zustands-Constraints, die die symbolischen Variablen beschreiben.

Daraus folgt, dass zwei symbolische Zustände potentiell dann gleich sind, wenn sie im konkreten Sinne gleich sind und sie die gleiche Menge der Szenariovariablen und Objektattribute mit symbolischen Werten belegt haben. Dies wird als Definition für die *Ähnlichkeit von symbolischen Zuständen* festgehalten.

Für die symbolische Gleichheit von Zuständen werden im Folgenden mehrere Möglichkeiten beschrieben. Dabei geht es ersten Abschnitt um vollständige Verfahren zur Zustandsvereinigung. Im zweiten Abschnitt geht es um Zustandsvereinigungen mit Hilfe von Approximationen. Der dritte Abschnitt zeigt die Möglichkeiten ohne symbolische Zustandsvereinigung.

4.12.1 Zustandsvereinigung mit Teilmengenvergleich

In diesem Abschnitt wird das Konzept für ein vollständiges Zustandsvereinigungsverfahren beschrieben. Die Vollständigkeit zielt dabei auf die Erkundung aller erreichbaren Zustände der Spezifikation ab. Dabei werden verschiedene symbolische Zustände auf Teilmengenbeziehungen geprüft. Dieses Konzept beschränkt sich auf den Test, ob ein neu erkundeter Zustand eine Teilmenge eines bereits erkundeten Zustandes ist. Bei erfolgreichem

Test kann der neu erkundete Zustand verworfen und die Transition auf den bereits erkundeten Zustand gesetzt werden. Den Test nur in dieser Richtung durchzuführen bringt den Vorteil, dass keine Zustände und Transitionen reevaluiert werden müssen. Der Nachteil ist ein etwas größerer Graph.

Der bei diesem Verfahren entstehende Zustandsgraph ist eine Überapproximation. Nicht alle Pfade, die in ihm zu finden sind, entsprechen einem konkreten Pfad. Es sind jedoch alle Zustände im Graphen erreichbar. Dies hängt mit der Vereinigung von Zuständen zusammen, die die Teilmengenbeziehung erfüllen. Dadurch führen Transitionen zu symbolischen Zuständen, die eine größere Menge an konkreten Zuständen und dadurch mehr Möglichkeiten repräsentieren als ursprünglich mit der Transition erkundet wurden. Dieses Mehr an Möglichkeiten bezieht sich auf die Parameterwertebereiche der ausgehenden Transitionen, die im Folgezustand Splitbedingungen erfüllen können.

Ein symbolischer Zustand repräsentiert eine Menge von konkreten Zuständen. Bei der Zustandsvereinigung muss überprüft werden, ob alle konkreten Zustände eines symbolischen Zustandes γ_a in einem anderen Zustand γ_b enthalten sind. Ist dies erfolgreich, wird der Zustand ersetzt, da alle Verhaltensmöglichkeiten von Zustand γ_a bereits in Zustand γ_b enthalten sind.

$$\gamma_a \subseteq \gamma_b$$

Um den Vergleich auf symbolischer Ebene durchzuführen, müssen die symbolischen Variablen betrachtet werden. Es gibt eine Menge an geteilten Variablen α, die in beiden Zuständen vorhanden sind. Dies sind die Szenariovariablen und Objektattribute, die in beiden Zuständen als symbolisch markiert sind. Des Weiteren gibt es eine Menge an nicht geteilten Variablen β, die nur in einem der beiden Zustände vorkommen. Diese Variablen sind symbolische Variablen, die die Szenariovariablen oder Objektattribute beschreiben. Sie hängen vom Pfad ab, der zu diesem Zustand führt. Diese Variablenmengen werden durch die jeweiligen Zustands-Constraints C_a und C_b beschrieben.

Um zu zeigen, dass ein Zustand γ_a eine Teilmenge eines anderen Zustandes γ_b ist, muss gezeigt werden, dass die Zustands-Constraints C_a des ersten Zustands die Zustands-Constraints C_b des zweiten Zustands unter der Berücksichtigung der Variablen α implizieren.

$$C_a \Rightarrow C_b \tag{4.1}$$

Dies wird in einer Formel überführt, die ein Solver lösen kann:

$$\forall\alpha : \exists\beta_a : C_a \Rightarrow \exists\beta_b : C_b \tag{4.2}$$

Mithilfe von Regeln der Prädikatenlogik kann die Formel so vereinfacht werden, dass ein Existenzquantor wegfällt:

$$\forall\alpha, \beta_a : C_a \Rightarrow \exists\beta_b : C_b \tag{4.3}$$

Für die Umsetzung des Konzepts wird der Z3 SMT-Solver verwendet (siehe dazu auch Abschnitt 4.1 und 5.3).

Betrachten wir folgendes Beispiel: Zwei Zustände werden auf die Teilmengenbeziehung geprüft. Diese besitzen eine gemeinsame Variable $param0$. Zudem besitzen die Zustände jeweils eine Menge nicht geteilter Variablen β. Beide Zustände haben Zustands-Constraints, wobei keine der Variablen ist an einen Wertebereich gebunden.

$$\begin{aligned} \alpha &= \{param0\}, \\ \beta_a &= \{sub, constVar\}, \\ \beta_b &= \{super, constVarSp\}, \\ C_a &= \{(param0 = sub), (sub = constVar * constVar)\}, \\ C_b &= \{(param0 = super), (super = constVarSp)\} \end{aligned}$$

Vereinfacht beschreibt die Gleichung, dass die Menge aller quadrierten genzen Zahlen eine Teilmenge der ganzen Zahlen ist:

$$\begin{aligned} X &= \{x | x = constVar^2\} \\ Y &= \{y | y = constVarSp\} \\ X &\subseteq Y \end{aligned}$$

Im Z3 Syntax lässt sich das Beispiel, in Form der Gleichung (4.3), folgendermaßen darstellen:

```
1  (assert
2    (forall ((sub Int) (constVar Int) (param0 Int))
3      (=>
4        (and
5          (= sub (* constVar constVar))
6          (= param0 sub))
7        (exists ((super Int) (constVarSp Int))
8          (and
9            (= super constVarSp)
10           (= param0 super))))))
11   (check-sat-using (then qe sat))
```

Listing 4.10: Teilmengenvergleich im Z3 Syntax

Formeln die alternierende Quantoren enthalten sind im Allgemeinen für Solver schwer zu lösen. Aus diesem Grund muss bei der Übergabe der Gleichung an den Solver auf die richtige Wahl der Taktik geachtet werden. Für Gleichungen wie sie beim Teilmengenvergleich entstehen, ist *Quantoren Elimination* [8] als Taktik zum Auflösen der Quantoren am besten geeignet (vergleiche Zeile 11 von Listings 4.10). Anschließend kann die Standard Taktik des Solvers verwendet werden.

4.12.2 Zustandsabstraktion

Durch die symbolische Ausführung mit Zustandsvereinigung mittels Teilmengenvergleich können Zustandsräume verkleinert werden. Trotzdem können immer noch sehr große Zustandsräume existieren.

Um schneller ein Ergebnis zu bekommen, kann es von Vorteil sein, die Spezifikation zuerst mit einer Form der Zustandsabstraktion zu prüfen, um schneller Hinweise auf mögliche Fehler in der Spezifikation zu bekommen. Dieses Verfahren kann als Schnelltest verstanden werden, mit dem die Spezifikation falsifiziert werden kann.

Die Vereinigung von *ähnlichen* Zuständen hat zur Folge, dass Informationen verloren gehen und dadurch einige Zustände nicht erkundet werden. Was zugleich den Zugewinn an Performance begründet Auf Grund der Tatsache, dass nicht alle Zustände erkundet werden, ist die Menge der Zustände nicht mehr vollständig.

Um eine Unterapproximation auszuführen, muss diese in der Konfiguration aktiviert werden. Dies ist in Listing 4.11 in Zeile 10 zu sehen.

```
1  import "TemperatureSensor.sml"
2  configure specification TemperatureSensor
3
4  use instancemodel "TemperatureSensorContainer.xmi"
5
6  symbolic parameters {
7    Controller.reportTemperatureSensor1(sensor1Value:symbolic),
8    Controller.reportTemperatureSensor2(sensor2Value:symbolic)
9  }
10 under-approximation: on
```

Listing 4.11: Konfiguration einer Unterapproximation

Bei der in dieser Arbeit verwendeten Abstraktion werden Zustände als gleich erkannt, wenn sie den gleichen Ausführungszustand besitzen und die selben Szenariovariablen und Objektattribute als symbolisch gekennzeichnet sind. Die Werte dieser Variablen und die Zustands-Constraints werden dabei nicht berücksichtigt.

Visser et al. [48] untersuchen vollständige und Verlustbehaftete Verfahren. Dabei kommen sie zu dem Schluss, dass symbolische Ausführung mit Teilmengenvergleich bei Metriken wie *predicate coverage* (Bedingungsüberdeckung) am besten funktioniert. Bei dieser Metrik müssen alle Bedingungen jeweils einmal zu *wahr* und zu *falsch* ausgewertet werden. Sie empfehlen jedoch auch verlustbehaftete Verfahren wie die *Shape Abstraction* (Strukturabstraktion). Dies ist sehr ähnlich zu der in dieser Arbeit verwendeten Unterapproximation. Da es weniger Laufzeit kostet, schlagen sie vor, dieses Verfahren zuerst zu verwenden, bevor die eher aufwendigen Verfahren angewendet werden.

Von Anand et al. [2] wird eine zweiteilige Zustandsvereinigungsprozedur beschrieben, die ähnlich zu der hier verwendeten ist. Dabei vergleichen Anand et al. zuerst die Struktur des Heaps und anschließend die symboli-

schen Werte mit Hilfe der Zustands-Constraints. Für eine Strukturabstraktion reicht es aus, wenn der erste Schritt der Zustandsvereinigung ausgeführt wird. Im Falle dieser Arbeit ist der erste Schritt mit der *Ähnlichkeit von symbolischen Zuständen* gleich zu setzten (siehe Abschnitt 4.12).

```
1  import "../mydomain.ecore"
2
3  system specification TemperatureSensor {
4
5  domain mydomain
6
7  define Controller as controllable
8  define Environment as uncontrollable
9
10 parameter ranges {
11   Controller.reportTemperatureSensor1(sensor1Value = [ 2 .. 5]),
12   Controller.reportTemperatureSensor2(sensor2Value = [ 2 .. 4 ])
13 }
14
15 collaboration TemperatureSensor {
16
17 dynamic role Controller ctr
18 dynamic role Environment env
19
20 specification scenario SensorAdapter {
21   var EInt temperature
22   alternative{
23     message env -> ctr.reportTemperatureSensor1(bind to temperature)
24     // TODO: temperature = ((temperature * 4) + 2) / 5
25   } or {
26     message env -> ctr.reportTemperatureSensor2(bind to temperature)
27   }
28   message strict requested ctr -> controller.reportTemperature(temperature)
29 }
30
31 specification scenario NormalizedBoundCheck {
32   var EInt temperature
33   message ctr -> controller.reportTemperature(bind to temperature)
34   violation if [ 2 > temperature | temperature > 4 ]
35   }
36 }
37 }
```

Listing 4.12: Spezifikation, die einen Adapter für verschiedene Sensornachrichten darstellt

Die Spezifikation in Listing 4.12 beschreibt eine Art Adapter für verschiedene Temperatursensoren, die ihre gemessene Temperatur mitteilen. Dabei wird das Szenario *SensorAdapter* mit einer Sensorspezifischen Nachricht initialisiert. Der Parameter der Nachricht wird anschließend von einer generelleren Systemnachricht übernommen und weiterverarbeitet. Wie in Zeile 11 und 12 zu sehen ist, haben die Nachrichten verschiedene Wertebereiche. Deswegen soll das Szenario zusätzlich die Aufgabe einer Normalisierung der Parameterwerte übernehmen. Diese ist allerdings nur als *TODO* in dem Szenario enthalten. Das Szenario *NormalizedBoundCheck* ist ein Systemtest, der diese Eigenschaft überprüfen soll.

Der Zustandsgraph in Fig. 4.2 zeigt den vollständigen und minimalen Zustandsgraphen der Spezifikation aus Listing 4.12, wie er von der symbolischen Ausführung mit Teilmengenvergleich erkundet wurde.

Für die Abstraktion ist es entscheidend in welcher Reihenfolge die

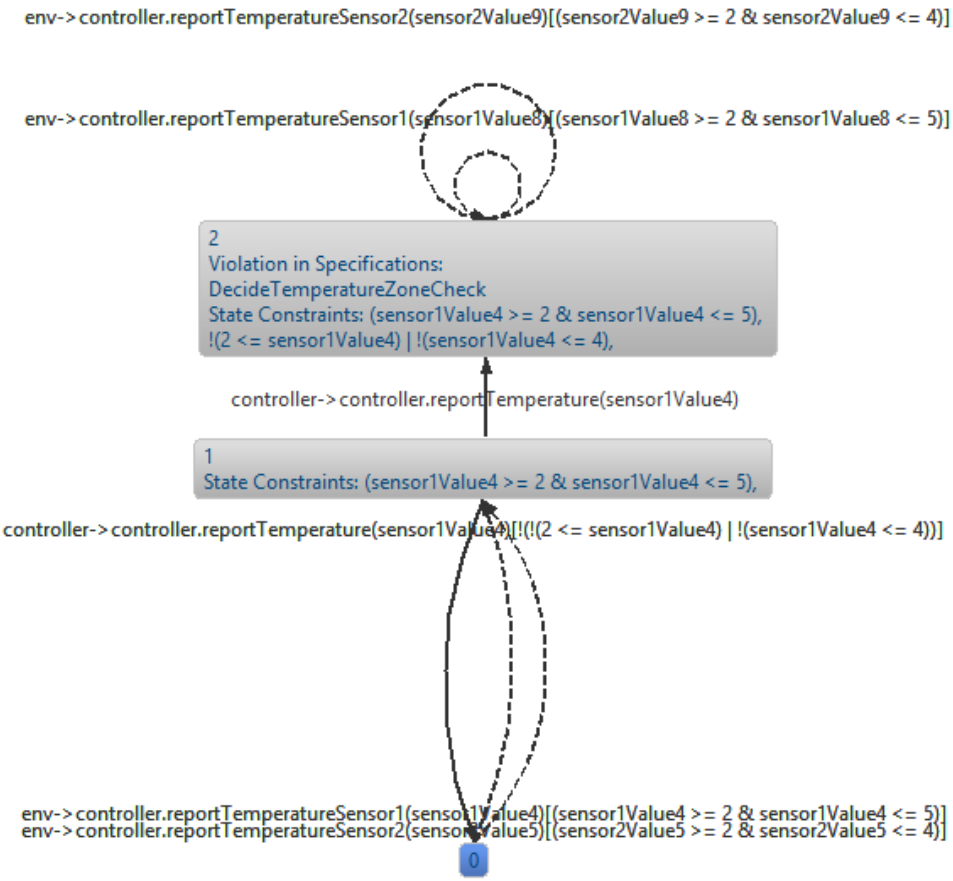

Abbildung 4.2: Vollständiger Zustandsgraph

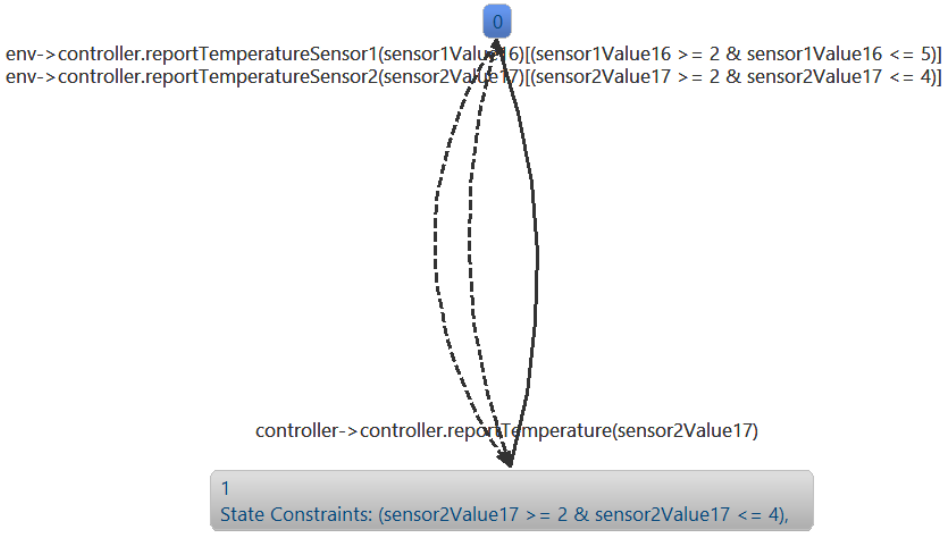

Abbildung 4.3: Unvollständiger Zustandsgraph durch Abstraktion

Nachrichten ausgeführt werden. Wird die Nachricht mit dem kleineren Wertebereich zuerst ausgeführt und anschließend die Nachricht mit dem größeren Wertebereich, kann diese mit dem vorher erkundeten Zustand vereint werden. Da die Verletzung nur bei Werten größer gleich 5 auftritt, bleibt sie in diesem Fall unerkannt. Der dabei entstehende Zustandsgraph ist in Abbildung 4.3 dargestellt. Ist die Ausführungsreihenfolge umgekehrt, entspricht der Zustandsgraph dem in Abbildung 4.2 dargestellten.

Die Unterapproximation weist ein großes Potential auf, Fehler schneller in Spezifikationen zu entdecken. Sie sollte jedoch nur zum Falsifizieren verwendet werden, da die erkundeten Zustände nicht vollständig sind.

4.12.3 Keine symbolische Zustandsvereinigung

Es kann vorkommen, dass Constraints beim Teilmengenvergleich nicht mehr effizient von Solver gelöst werden können. Dann kann das Vereinigen von symbolischen Zuständen ausgeschaltet werden. Es werden dann nur Zustände gleich gesetzt, die auf Basis eines String-Vergleichs die gleichen Constraints und die gleichen symbolischen Variablen besitzen. Der Zustandsgraph bleibt dadurch theoretisch vollständig.

Allerdings kann es zu unendlichen Ausführungsbäumen oder Graphen kommen. Es entsteht ein Graph, da konkrete Zustände sowie symbolische Zustände, die exakt die gleichen Zustands-Constraints besitzen, vereinigt werden können. Unendliche Ausführungsbäume entstehen in der symbolischen Ausführung, wenn das getestete Programm Schleifen enthält. Diese Schleifen sind in SML-Spezifikationen nicht eindeutig zu erkennen. In einem

konkreten Zustandsgraphen können sie hingegen an Schlaufen im Graphen erkannt werden.

Um unendliche Ausführungsbäume in den Griff zu bekommen, muss der Explorationsalgorithmus angepasst werden. Dieser muss eine Abbruchbedingung unterstützen, sodass er bei einer gegebenen Suchtiefe die Erkundung beendet. Für diese Aufgabe gibt es eine Variante der *Iterativen Tiefensuche*. Diese wird solange ausgeführt bis eine Sicherheitsverletzung gefunden wird oder die maximale Suchtiefe erreicht wurde.

Die maximale Suchtiefe bestimmt dabei wie nahe die Exploration an eine vollständige Exploration heran geht. Dieser Wert muss für jede SML-Spezifikation individuell bestimmt werden. Dabei entsprechen alle Pfade, die während der Exploration erkundet werden, konkreten Pfaden.

Um die symbolische Zustandsvereinigung zu deaktivieren, muss dies in der Konfiguration angegeben werden, wie es in Zeile 10 von Listing 4.11 zu sehen ist.

```
1  import "TemperatureSensor.sml"
2  configure specification TemperatureSensor
3
4  use instancemodel "TemperatureSensorContainer.xmi"
5
6  symbolic parameters {
7    Controller.reportTemperatureSensor1(sensor1Value:symbolic),
8    Controller.reportTemperatureSensor2(sensor2Value:symbolic)
9  }
10 symbolic-state-matching: off
```

Listing 4.13: symbolische Zustandsvereinigung deaktivierende Konfiguration

4.13 Konkrete Testfälle aus symbolischer Analyse

Aus symbolisch explorierten Pfaden können explizite Pfade extrahiert werden. Diese Pfade können anschließend als Testfälle benutzt werden.

Für die Bearbeitung von SML-Spezifikationen wurde in dieser Arbeit eine Methode erstellt, die symbolische Pfade zu einer Sicherheitsverletzung in einen konkreten Pfad umwandeln kann. Dabei sind generell alle Explorationsverfahren dazu ausgelegt Sicherheitsverletzungen zu finden. Demnach können aus allen Verfahren konkrete Pfade gewonnen werden.

Bei den Verfahren der symbolischen Ausführung mit Teilmengenvergleich und der Unterapproximation gilt dabei jedoch zu beachten, dass nicht alle Pfade die im Zustandsgraphen zu finden sind einem konkreten Pfad entsprechen. Um diesen Fall bei der Erstellung eines konkreten Pfades zu vermeiden, kann eine Technik wie das *counterexample guided abstraction refinement (CEGAR)* [13] angewendet werden. Dabei wird aus dem symbolischen Zustandsgraphen ein Pfad ausgewählt, der zu einer Sicherheitsverletzung führt, und dieser in der konkreten Ausführung getestet. Ist der Pfad unecht, wird er im Zustandsgraph als solcher gekennzeichnet. Dies wird solange fortgesetzt bis ein echter Pfad zu einer Sicherheitsverletzung gefunden wurde.

In dieser Arbeit wurde zur Erstellung von expliziten Pfaden aus Zustandsgraphen die Exploration mit deaktivierter symbolischer Zustandsvereinigung gewählt. Der Vorteil an diesem Verfahren ist, dass es hierbei keine unechten Pfade gibt. Demnach kann jeder symbolische Pfad in einen expliziten Pfad umgewandelt werden.

Bei der Erstellung von expliziten Pfaden müssen die konkreten Nachrichten, die auf dem Pfad liegen, ermittelt werden. Dazu müssen die Werte des symbolischen Parameters ermittelt werden. Um dies zu erreichen, werden alle Zustands-Constraints der Zustände, die sich auf dem Pfad befinden, an den symbolischen Interpreter übergeben. Zudem werden alle symbolischen Parameter an den symbolischen Interpreter übergeben. Zu diesen muss nun eine konkrete Lösung gefunden werden. Diese Aufgabe übernimmt der SMT-Solver. Ist dieser erfolgreich, können die konkreten Werte der symbolischen Parameter aus dem Modell abgelesen werden. Auf diese Weise können auch konkrete Werte für symbolische Objektattribute aus dem Startzustand ermittelt werden.

Die Variablen und Constraints können auf einem Pfad nur eingeschränkt werden. Dadurch können sich die Zustands-Constraints der verschiedenen Zustände gegenseitig nicht widersprechen. Jedoch ist es wichtig, alle Zustands-Constraints mit einzubeziehen, da bei der Entwicklung von Pfaden Variablen aus den Zuständen verschwinden können. Wenn Variablen keinen direkten oder indirekten Einfluss auf die aktuellen Variablen oder Zustands-Constraints mehr haben, werden diese entfernt. Ein indirekter Einfluss ist gegeben, wenn die Variable in einem Constraint vorkommt, das eine aktuelle Variable beschreibt.

4.14 Anwendungsszenario

Spezifikationen können symbolisch ausgeführt werden. Dies hilft bei Spezifikationen, die Integer mit einem großen Wertebereichen als Eingabe aus der Umwelt bekommen. Diese Spezifikationen sind in den meisten Fällen in der expliziten Ausführung nicht bezwingbar, da sie unter der Zustandsraumexplosion leiden. Die symbolische Ausführung ermöglicht hingegen die praktische Erkundung diese Spezifikationen bis zu einer gewissen Komplexität.

Dazu kann eine SML-Spezifikation in ScenarioTools entwickelt werden, welche anschließend mit einem Schnelltest (Unterapproximation) exploriert werden kann. Wird dabei ein Fehler gefunden, kann dieser im *StateGraph View* betrachtet werden. Dem Benutzer wird hierbei der Pfad zur Verletzung farblich markiert. Zusätzlich kann ein expliziter Pfad zur Sicherheitsverletzung erstellt werden. An diesem kann der Fehler zusätzlich im expliziten *Play-Out-Modus* nachvollzogen werden. Dies ist besonders wichtig, weil die symbolische Ausführung teilweise sehr abstrakte Ergebnisse

liefert. Sobald der Benutzer eine Lösung gefunden hat, die den Fehler vermeidet, kann er die vorherigen Schritte wiederholen bis der Schnelltest keine Sicherheitsverletzung mehr findet.

Als zweiten Schritt kann nun die Exploration mit Teilmengenvergleich ausgeführt werden. Wird hierbei ein Fehler gefunden, kann wie im vorherigen Schritt (inklusive des Schnelltests) verfahren werden. Diese Prozedur muss solange durchgeführt werden, bis die Exploration mit Teilmengenvergleich keinen Fehler mehr findet.

4.15 Explorationsalgorithmus

Zur Exploration des Zustandsraums stehen in ScenarioTools bereits einige Verfahren für explizite SML-Spezifikationen zur Verfügung. In diesem Abschnitt wird geprüft, wie diese Verfahren für die symbolische Zustandsraumerkundung verwendet werden können.

Im Grunde gibt es zwei wesentliche Verfahren. Das eine Verfahren verwendet die Tiefensuche, das andere ist ein Syntheseverfahren, das *On-The-Fly* einen Controller für ein System erstellen kann.

Bei der Analyse hat sich heraus gestellt, dass die Synthese nicht direkt anwendbar ist. Dies begründet sich zum einen darin, dass dieses Verfahren hauptsächlich auf abstrakter Ebene der Zustände und Transitionen arbeitet und den Zustandsgraphen analysiert. Dieses Verfahren müsste dabei berücksichtigen, dass bei der Exploration mit Teilmengenvergleich ungültige Pfade entstehen. Hier gibt es Verfahren wie *counterexample guided abstraction refinement (CEGAR)* [13], die versuchen dieses Problem in den Griff zu bekommen. Katz [29] hat dies bereits erfolgreich an *Behavioral Programming* (eine Form der szenariobasierten Entwicklung) umgesetzt. Allerdings wurde dabei eine andere Approximation gewählt. Zudem stand die Behandlung von symbolischen Parametern nicht im Mittelpunkt.

Ein weiteres Problem stellt das Verfahren zum Erstellen eines Controllers dar. Dabei wird in einem zwei Spieler Spiel ein *Gamegraph* erstellt [21]. Hierbei kann sich das System während eines Systemschrittes für eine Transition entscheiden. Während eines Umweltschrittes müssen alle Transitionen verfolgt werden. Auf diese Weise kann ein Teilgraph ermittelt werden, der die SML-Spezifikation erfüllt. Wobei der Teilgraph keine Sicherheitsverletzung enthalten darf. Da ein Zustand nun eine Menge an Zuständen repräsentiert, sind ausgehende Transitionen von den konkreten Werten ihrer Parameter abhängig. Dabei kann eine Nachricht mit einem Parameterwert über einem gewissen Schwellenwert einen anderen Nachfolgezustand haben als die gleiche Nachricht mit einem Parameterwert unter diesem Schwellenwert.

Die Tiefensuche kann mit kleinen Anpassungen für die symbolische Zustandsraumerkundung genutzt werden. Diese eignet sich auch besser für Vergleiche zwischen verschiedenen Zustandsvereinigungsverfahren, da sie

den ganzen Zustandsraum erkundet. Die *On-The-Fly-Synthese* hingegen erkundet den Zustandsraum nur so weit bis sie in der Lage ist, einen gültigen Controller zu erstellen. Zudem arbeitet die Tiefensuche deterministisch, vorausgesetzt die Zustände geben ausgehende Transitionen immer in der gleichen Reihenfolge an die Tiefensuche weiter. Da die Ausführungslogik im Grunde auch deterministisch arbeitet, ist bei der Tiefensuche immer das gleiche Ergebnis zu erwarten solange sich die Spezifikation nicht ändert. Dies ist für Testzwecke wichtig, da die Erkundungsreihenfolge die Vereinigung von Zuständen beeinflussen kann.

4.16 Vereinigung von Nachrichten

Während der Ausführung einer symbolischen Spezifikation können explizite und symbolische Elemente vermischt werden. Aus diesem Grund hat es sich als vorteilhaft erwiesen, nach Beendigung des Sammelns von Nachrichten (siehe Abbildung 3.2) diese Nachrichten auf doppelte Vorkommnisse zu überprüfen. Dabei wird zum einen geprüft, ob es mehrere Nachrichten gibt, die die gleiche konkrete Nachricht repräsentieren. Zum anderen, ob es symbolische Nachrichten gibt, die jeweils die gleiche Menge an konkreten Nachrichten repräsentieren oder in einer Teilmengenbeziehung stehen. Diese Nachrichten können zusammengefasst werden. Auf diese Weise kann der Zustandsgraph minimal gehalten werden.

Auf Grund des umgesetzten Konzeptes ist es nicht möglich, konkrete Zustände mit symbolischen Zuständen zu vereinen. Durch das Vereinen von konkreten und symbolischen Nachrichten kann jedoch ein ähnlicher Effekt erzielt werden. Zudem wird durch das Zusammenfassen von Nachrichten die Anzahl der erkundeten Zustände minimiert. Dadurch werden zudem weniger der sehr aufwendigen Teilmengenvergleiche von symbolischen Zuständen durchgeführt.

4.17 Modellierung und Analyse von Zeit

Die Simulation von Zeit ist ein weiterer Aspekt, der sich gut mit der symbolischen Ausführung beherrschen lässt. Da die Modellierung von Zeit Thema einer anderen Arbeit ist, wird dieser Aspekt hier nur kurz thematisiert. Einige Konzepte, die für die symbolische Ausführung genutzt werden, können auch dazu genutzt werden, Zeit zu modellieren. Dabei kann das Verstreichen von Zeit als Umweltnachricht dargestellt werden. Tritt diese Nachricht auf, erhöht das System eine symbolische Zeit-Variable um eine Einheit. Dieses Verhalten ist in Szenario *Time* in Listing 4.14 zu sehen. Dies kann auch im Zusammenhang mit der Annahme betrachtet werden, dass das System mit beliebig, jedoch endlich vielen Systemnachrichten auf eine Umweltnachricht antworten kann bevor die Umwelt die nächste Nachricht sendet. Diese

Annahme beschreibt in anderen Worten, dass das System beliebig schnell gegenüber der Umwelt ist. Mit dieser Art der Modellierung kann die Umwelt dafür sorgen, dass Zeit vergeht.

Das Szenario *OvenRegulation* beschreibt den Ablauf zwischen der Eingabe einer neu gemessenen Temperatur und des Ein- oder Ausschaltens des Heizers unter der Berücksichtigung von Zeit. Nach der Eingabe einer neu gemessenen Temperatur wird zwei Zeiteinheiten gewartet, um beispielsweise den Heizer in einem sinnvollen Intervall ein oder aus zuschalten. Um Zeit verstreichen zu lassen, sendet die Umwelt die *tick()*-Nachricht. Ist die Zeit verstrichen, kann der Befehl an den Heizer gesendet werden. Anschließend wird geprüft, ob das Szenario unter einer bestimmten Zeitschranke geblieben ist. Da die Umwelt zwischen dem Ablaufen der *wait-until*-Bedingung und der letzten Bedingung nicht die Chance hatte weitere Zeit verstreichen zu lassen, kann dieses Szenario erfolgreich beendet werden.

Durch die Einkommentierung der Nachricht in Zeile 19 wird das System dazu gezwungen, auf die Umweltnachricht *selfCheck()* zu warten. Nun hat die Umwelt die Chance beliebig viel Zeit verstreichen zu lassen bevor diese Nachricht auftritt. In dieser Variante ist das Szenario nicht erfüllbar.

Eine erweitertes Konzept für die Modellierung von Zeit wird von Wang [49] im Kontext von *LSC* vorgestellt. Dabei können verschiedene Bedingungen an die Zeit gestellt werden. Diese werden im Laufe der Ausführung gesammelt und in Relation gesetzt. Die Zeit wird als symbolische Variable betrachtet. Der Wert dieser Variable kann zwischengespeichert und als Referenzwert genutzt werden. Eine Bedingung beschreibt nun wie viel Zeit minimal oder maximal vergangen sein darf in Form von Ungleichungen wie beispielsweise $Zeit < t_1 + 3$. t_1 ist dabei ein Zeitstempel zu einem früheren Zeitpunkt. $Zeit$ ist die aktuelle Zeit. Bei der Ausführung werden die verschiedenen Zeit-Bedingungen bezüglich ihres Auftretens in eine kleinergleich Relation gesetzt. Die Zeit-Variable aus der ersten Bedingung wird somit zu $Zeit_0$. Die aktuelle Zeit aus der zweiten Bedingung wird als $Zeit_1$ dargestellt. Diese neuen Variablen werden durch folgendes Constraint beschrieben: $Zeit_0 \leq Zeit_1$.

Um dieses Konzept in ScenarioTools zu übernehmen, müsste zusätzlich eine neue Zeitbedingung hinzugefügt werden. Durch diese können die Bedingungen an die Zeit gestellt werden. Die Auswertung dieser Bedingungen erfolgt nicht wie die Auswertung anderer symbolischer Bedingungen. Hier wird nur geprüft, ob die Bedingung in Bezug auf die Zeit-Constraints erfüllbar ist. Wenn die Zeit-Constraints nicht erfüllt werden können, ist der Ausführungspfad nicht gültig. Demnach gibt es keine Splitbedingung bei Zeit-Constraints.

Offen bleibt, welchen Einfluss die Zustandsvereinigung bei der Umsetzung des von Wang beschriebenen Verfahrens auf die Simulation von Zeit hat. Zu klären wäre hier, ob eine Zustandsvereinigung erlaubt ist und wenn ja, unter welchen Bedingungen dies gilt. Weitere zu klärende Fragen sind:

Ist es ausreichend, wenn die Zeit-Constraints erfüllt sind oder müssen diese identisch sein?

```
1  collaboration OvenCollaboration {
2
3    static role Controller ctr
4    static role TemperatureSensor ts
5    static role Heater heater
6    static role Clock clock
7
8    singular specification scenario OvenRegulation {
9      var EInt tmp
10     message ts -> ctr.reportTemperature(bind to tmp)
11     var EInt t = clock.time
12     wait until [ clock.time >= t + 2]
13     alternative if [tmp > ctr.nominalTemperature]{
14       message strict requested ctr -> heater.turnOff()
15     } or if [ tmp <= ctr.nominalTemperature]{
16       message strict requested ctr -> heater.turnOn()
17     }
18     // Das System muss auf eine Umweltnachricht warten, dadurch
19     // bekommt die Umwelt die Möglichkeit die Zeitbedingugn zu verletzen.
20 //  message strict heater -> heater.selfCheck()
21     violation if [clock.time > t + 3]
22   } constraints [ ignore message ts -> ctr.reportTemperature(*) ]
23
24   specification scenario Time {
25     message clock -> clock.tick()
26     var EInt time = clock.time + 1
27     message strict requested ctr -> clock.setTime(time)
28   }
29 } // end Collaboration
```

Listing 4.14: Beispiel für eine Spezifikation, die diskrete Zeit berücksichtigt

Kapitel 5

Implementierung

In diesem Kapitel wird die Implementierung der symbolischen Ausführung als Erweiterung von ScenarioTools beschrieben. Dabei werden auf einige Veränderungen und Neuerungen hingewiesen sowie technische Details der Implementierung geklärt.

5.1 Neue Plugins

Um die symbolische Ausführung in ScenarioTools zu realisieren, wurden mehrere neue Plugins erstellt. Bei der Erstellung von Plugins musste auf die gegebene Struktur in ScenarioTools sowie auf das Abhängigkeitsmodell von Eclipse Plugins geachtet werden. Dies begründet die große Anzahl an Plugins, welche nachfolgend beschrieben werden.

- org.scenariotools.sml.runtime.builder

  Das *Builder-Plugin* abstrahiert die Erstellung von Zuständen und Zustandsgraphen. Je nach Konfiguration der Ausführung kann der *Builder* entscheiden, ob explizite oder symbolische Zustände erstellt werden.

- org.scenariotools.sml.runtime.factory

  Das *Factory-Plugin* stellt *Factories* für Zustände, Zustandsgraphen und Szenarios bereit.

- org.scenariotools.sml.runtime.types

  Das *Types-Plugin* enthält eine Aufzählung von verschiedenen Ausführungstypen, die von ScenarioTools unterstützt werden. Aktuell umfasst es die beiden Ausführungstypen *default* (bzw. explizit) und *symbolisch*.

- org.scenariotools.sml.runtime.symbolic

Das *symbolic-Plugin* erweitert die explizite Ausführungslogik. Dabei wurden die Konzepte für eine symbolische Ausführung angepasst. Dazu zählt zum einen die Ersetzung des Interpreters für konkrete Ausdrücke durch einen symbolischen Interpreter. Zu anderen wurde die Verarbeitung der Ergebnisse des Interpreters an die Bedürfnisse einer symbolischen Ausführung angepasst.

- org.scenariotools.sml.runtime.symbolic.constraints

  Das *Constraint-Plugin* enthält Wrapperklassen für symbolische Constraints und symbolische Werte. Dadurch ist die Ausführungslogik unabhängig von einem speziellen SMT-Solver.

- org.scenariotools.sml.runtime.symbolic.events

  Das *symbolic-events-Plugin* erweitert die konkreten Nachrichten um die benötigten symbolischen Eigenschaften.

- org.scenariotools.sml.runtime.symbolic.interpreter

  Das *symbolic-Interpreter-Plugin* enthält die Schnittstelle zum *Z3 SMT-Solver*. Hier werden symbolische Constraints gelöst, Teilmengentests von Zuständen durchgeführt oder Bedingungen auf Erfüllbarkeit geprüft.

5.2 Wichtige Erweiterungen von Metamodellen

In diesem Abschnitt werden die wichtigsten Erweiterungen der verschiedenen Metamodelle von ScenarioTools erklärt.

5.2.1 Konfiguration

Die Konfiguration der Ausführung bestimmt, ob eine Spezifikation symbolisch ausgeführt wird. Dazu wurde dem Metamodell (siehe Abbildung 5.1) je eine Liste für symbolische Parameter von Nachrichten und eine für symbolische Objektattribute hinzugefügt. Zudem gibt es zwei Wahrheitswerte, die bestimmen ob eine Unterapproximation oder eine Zustandsraumerkundung ohne Zustandsvereinigung durchgeführt werden soll. Da die Verfahren gegensätzliche Strategien bei der Zustandsvereinigung anwenden, wird von der Grammatik gewährleistet, dass nur einer der beiden Wahrheitswerte auf *wahr* stehen kann.

5.2.2 Symbolische Ausführungslogik

Das Metamodell der Ausführungslogik wurde um diejenigen Konzepte erweitert, die für eine symbolische Ausführung wichtig sind. Die wichtigsten dieser Konzepte sind in Abbildung 5.2 dargestellt. Dazu zählt ein Konzept zur

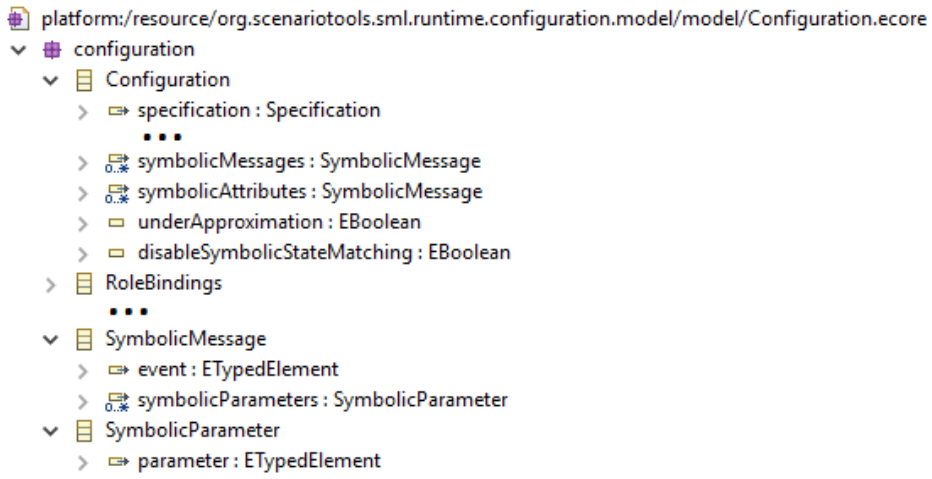

Abbildung 5.1: Metamodell der Konfiguration

Repräsentation von unerkundeten Programmpfaden. Dies wird in *UnexploredStateConstraints* abgebildet. Ein weiteres Konzept stellen die Zustands-Constraints (*StateConstraints*) dar. Diese enthalten zusätzlich eine Liste aller symbolischen Variablen eines Zustands. Der symbolische Zustandsgraph (*SymbolicStateGraph*) übernimmt Aufgeben wie das Bereithalten der unerkundeten Programmpfade. Die Ausführung dieser Programmpfade wird von diesem Punkt aus gesteuert. Das Konzept des symbolischen Zustands (*SymbolicState*) erweitert den expliziten Zustand um Zustands-Constraints.

5.2.3 Constraints und symbolische Variablen

Die Constraints und symbolischen Variablen sind durch Klassen abstrahiert. Dadurch ist die Ausführungslogik von einem bestimmten SMT-Solver unabhängig. Für die Ausgabe in der Benutzeroberfläche wird das Constraint und die symbolische Variable zusätzlich als Zeichenkette gespeichert. Das dazugehörige Metamodell ist in Abbildung 5.3 zu sehen.

5.2.4 Symbolische Nachrichten

Das Metamodell in Abbildung 5.4 erweitert explizite Nachricht um symbolische Konzepte. Zum einen kann eine symbolische Nachricht Constraints speichern. Dadurch können gleiche Nachrichten mit unterschiedlichen Constraints dargestellt werden. Zum anderen kann der Integer-Parameter symbolische Werte darstellen.

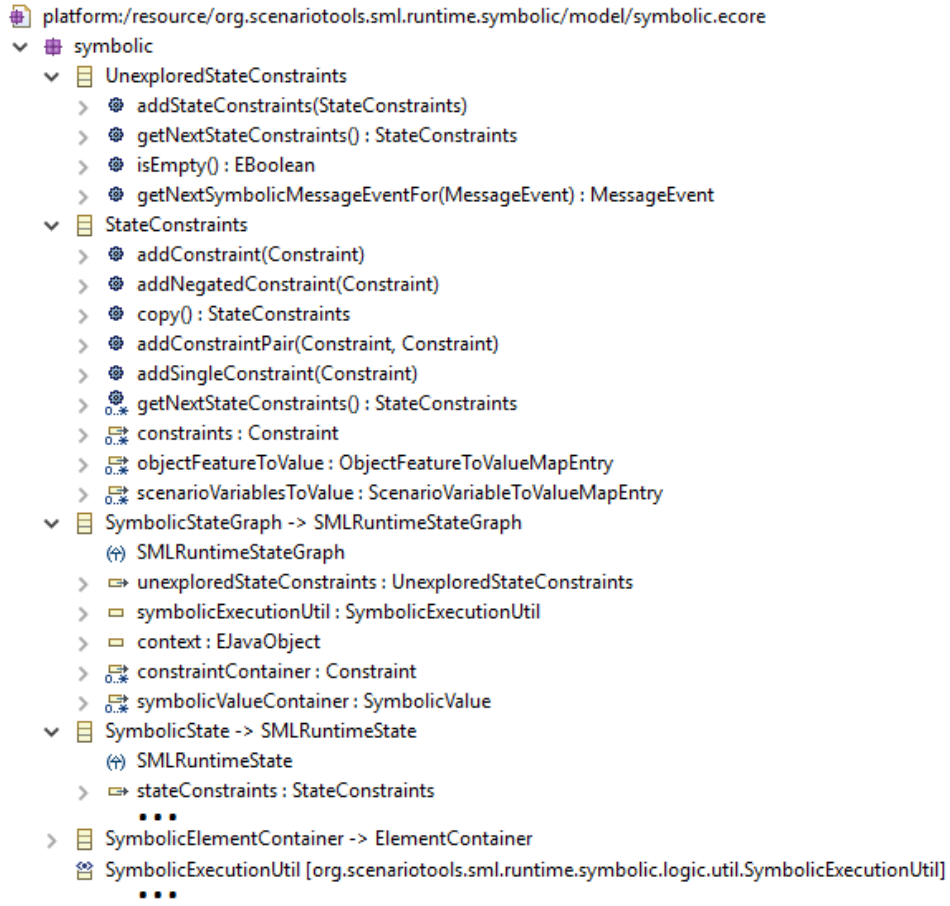

Abbildung 5.2: Metamodell-Erweiterung der Ausführungslogik

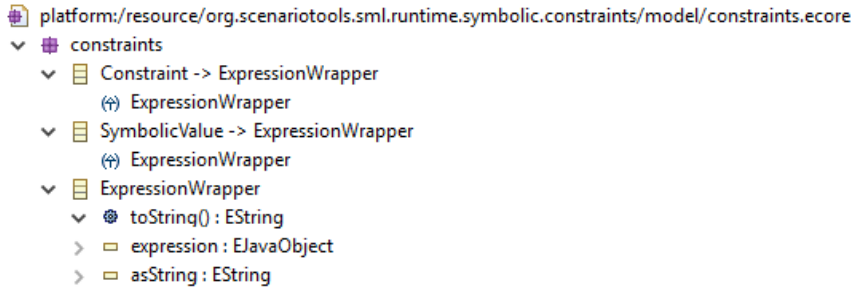

Abbildung 5.3: Metamodell der Constraints

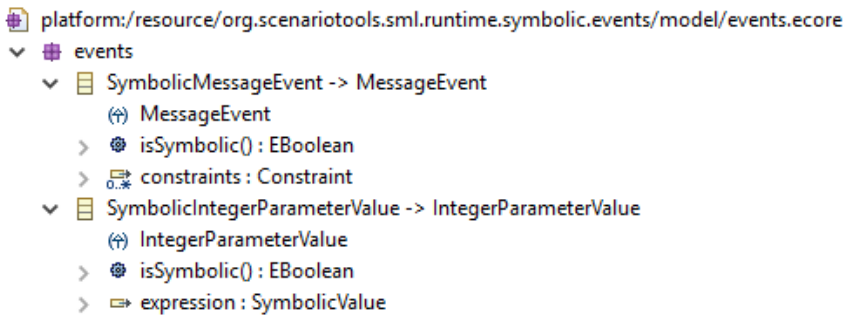

Abbildung 5.4: Metamodell-Erweiterung der Nachrichten

5.3 Auswahl des SMT-Solvers für den symbolischen Interpreter

Bei der Implementierung des symbolischen Interpreters wird der *Z3 SMT-Solver* als Backend verwendet. Nach einer Vorauswahl wurden folgende Solver genauer für einen Einsatz in ScenarioTools untersucht: *CVC4* [4], *Choco Solver* [39], und *Z3* [15]. Diese Solver haben aktuell die aktivsten Entwicklungs- und Nutzergemeinschaften.

Ein Kriterium für die Wahl ist die Bereitstellung einer Java API, da ScenarioTools in Java programmiert ist. Dies ist bei Z3 der Fall. Der Choco Solver ist ebenfalls in Java implementiert, sodass das Kriterium ebenfalls erfüllt ist. Bei CVC4 gibt es standardmäßig keine Java API. Sie ist nur verfügbar, wenn das Programm selbst mit den passenden Parametern kompiliert wird. Da Z3 und CVC4 nicht nativ in Java programmiert sind, bedeutet die Kommunikation über eine Java API Performanceverluste. Der Choco Solver hingegen kann direkt als Bibliothek eingebunden werden.

Ein weiteres Kriterium ist, dass der Solver gut mit Quantoren umgehen können soll. Diese Eigenschaft wird bei der Teilmengenbestimmung von Zuständen benötigt (siehe Abschnitt 4.12). Während der Umsetzung hat sich zudem herausgestellt, dass es von Vorteil ist, wenn der Solver *Quantoren Elimination* unterstützt. Dies ist eine spezielle Vorgehensweise beim Lösen von Formeln mit Quantoren. Für einen genauere Beschreibung siehe Kapitel 9.2 in [32]. Einige der getesteten Formeln ließen sich ohne *Quantoren Elimination* nicht lösen. CVC4 und Z3 beherrschen diese Technik. Der Choco Solver kann hingegen keine Quantoren handhaben.

Zudem ist die Handhabung von linearer Integer-Arithmetik und nicht-linearer Integer-Arithmetik wichtig, da dies die hauptsächlich in ScenarioTools gebrauchten Theorien sind. Dies bezüglich steht Z3 am besten dar [28] . In einem Vergleich von Felbinger et al. [18] schneidet Z3 ebenfalls als bester Solver ab.

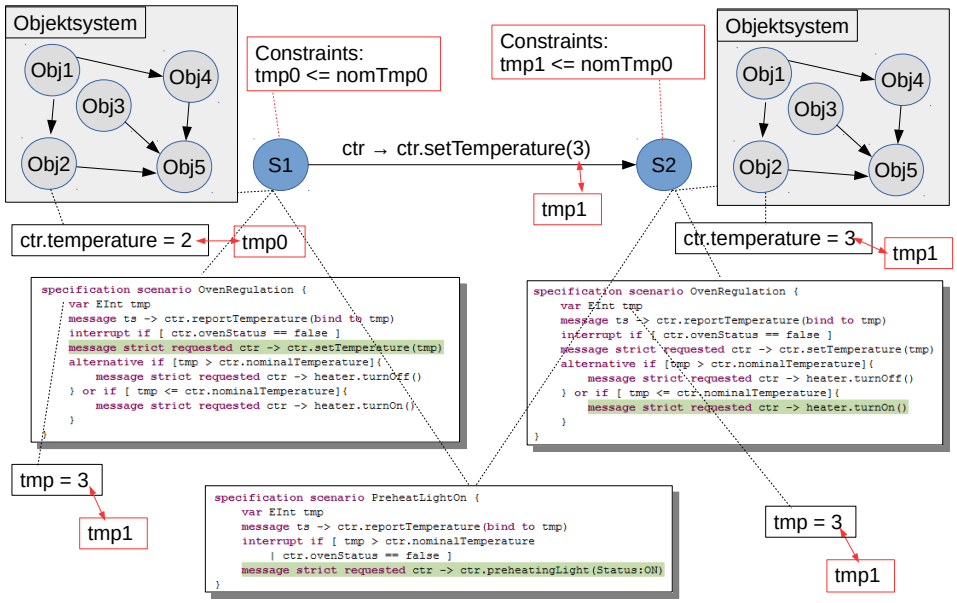

Abbildung 5.5: Elementcontainer Konzept mit Erweiterungen für die symbolische Ausführung (in rot)

Auf Grundlage dieser Erkenntnisse ist die Wahl auf *Z3* gefallen. Er schneidet bei den meisten Punkten am besten ab und unterstützt alle benötigten Eigenschaften.

5.4 Aufgabenbereiche des symbolischen Interpreters

Die Aufgaben des symbolischen Interpreters werden in drei Aufgabenbereiche unterteilt. Der erste Aufgabenbereich ist für die Entscheidung von Bedingungen verantwortlich. Die Parameterunifizierung symbolischer Nachrichten wird vom zweiten Aufgabenbereich übernommen. Der dritte Aufgabenbereich führt den Teilmengenvergleich von symbolischen Zuständen durch. Dabei werden die benötigten Konzepte als Parameter übergeben. Diese werden von den einzelnen Einheiten des symbolischen Interpreters in Gleichungen für den SMT-Solver umgewandelt. Das Ergebnis des SMT-Solvers wird anschließend von den einzelnen Einheiten interpretiert und in Anweisungen für die Ausführungslogik umgewandelt.

5.5 Zustandsrepräsentation im Zustandsgraphen

In der expliziten Variante von ScenarioTools ist die Repräsentation des Zustandsgraphen in Bezug auf den Speicherplatz optimiert. Dabei wird ein neu explorierter Zustand in seine Bestandteile zerlegt und diese anschließend in einen Elementcontainer eingefügt. Ist das Element bereits im Elementcontainer vorhanden, wird es gelöscht und alle Referenzen werden auf das bereits vorhandene Element umgesetzt. Können alle Elemente in einem Zustand ersetzt werden, kann letzten Endes der gesamte Zustand ersetzt werden. Ist dies der Fall, dann wurde ein Zustand zum wiederholten Mal erkundet. Im Zustandsgraph ist diese Situation durch eine Schlaufe zu erkennen. Dieses Verfahren ist möglich, da die einzelnen Elemente voneinander *unabhängig* sind. In Abbildung 5.5 ist dieses Verhalten anhand zweier Zustände dargestellt. Die rot umrandeten Elemente gehören *nicht* zum expliziten Konzept. Der Zustand *S1* besteht aus einem Objektsystem in dem das *obj2* die Rolle des Controllers übernimmt. Die aktuell gespeicherte Temperatur beträgt 2 Grad Celsius. Zudem besteht der Zustand aus zwei Szenarien: *OvenRegulation* und *PreheatLightOn*. Wobei die Variable *tmp* des ersten Szenarios den Wert 3 besitzt. Mit der Ausführung der Nachricht *ctr -> ctr.setTemperature(3)* folgt Zustand *S2*. Die *set*-Nachricht verändert die Temperaturvariable des Controllers. Damit stellt das Objektsystem eine neue Konfiguration dar. Ebenso verändert sich das Szenario *OvenRegulation*. Das Szenario *PreheatLightOn* hingegen bleibt unverändert. Dies wird nur einmal im Elementcontainer gespeichert. Die Referenzen beider Zustände zeigen auf dasselbe Szenario.

Dieses Konzept muss für die symbolische Ausführung erweitert werden. Das bringt einige Herausforderungen mit sich: Die eingangs geforderte Bedingung der Unabhängigkeit kann nicht mehr erfüllt werden. Das Konzept der symbolischen Ausführung sieht vor, dass konkrete Variablen durch symbolische Variablen ersetzt werden, welche wiederum durch Zustands-Constraints beschrieben werden. Dadurch besteht nun eine Abhängigkeit zwischen den Szenarien und dem Zustand selbst. Zudem repräsentiert ein symbolischer Zustand eine Menge konkreter Zustände. Die in Abbildung 5.5 rot umrandeten Elemente stellen diese Veränderung dar.

Um die Unabhängigkeit wiederherzustellen, wurde die Aufgabe des Elementcontainers in zwei Phasen unterteilt. Zudem wurden die Abhängigkeiten der symbolischen Variablen zum Zustand verschoben. Alle Informationen über symbolische Variablen in Szenarien und Objektsystemen werden auf Ebene der Zustands-Constraints gespeichert. In einem Szenario oder einem Objekt selbst wird lediglich ein Vermerk gespeichert, dass es sich um eine symbolische Variable handelt. In der ersten Phase kann der Elementcontainer auf Grundlage von konkreten Werten und symbolischen Markern entscheiden, ob zwei Elemente eines Zustandes identisch sind und diese dann ersetzen. In der zweiten Phase wird der Zustand selbst

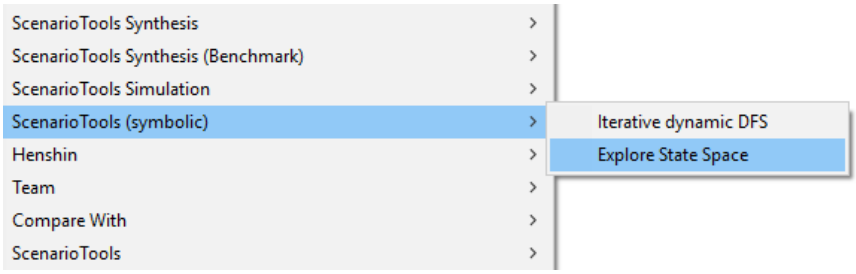

Abbildung 5.6: Kontextmenü in ScenarioTools

betrachtet. Hier werden die symbolischen Variablen in die Szenarien und in das Objektsystem eingesetzt. Anschließend wird mit Hilfe der Zustands-Constraints ein Teilmengenvergleichen durchgeführt.

Um die Abhängigkeiten von Szenarien und Zuständen vollständig aufzulösen, muss zudem beachtet werden, dass Nachrichtenparameter von symbolischen Variablen abhängig sein können. Diese Abhängigkeiten konnten bis aus eine Ausnahme aufgelöst werden. Die Ausnahme bilden die *Inkrement*-Ausdrücke in *set*-Nachrichten. Diese Nachrichten verändern das Objektsystem. Die Änderung wird während eines Ausführungsschrittes als erstes umgesetzt. Soll die Nachricht anschließend in einem Szenario unifiziert werden, muss ihr Parameter ausgewertet werden. Dies führt dazu, dass der bereits veränderte Wert aus dem Objektsystem ausgelesen und erneut verändert wird. Dadurch wird die Unifizierung unmöglich. Eine solche Nachricht ist in Listing 5.1 zu sehen. Um dieses Problem zu umgehen, kann die Berechnung des Parameters in eine Variable ausgelagert werden. Dieses Vorgehen ist in Punkt 2 in Listing 5.1 beschrieben.

```
1 // 1) funktioniert nicht
2 message ctr -> ctr.setTemperature(ctr.temperature + 1)
3 // 2) funktioniert
4 var EInt tmp = ctr.temperature + 1
5 message ctr -> ctr.setTemperature(tmp)
6 }
```

Listing 5.1: Problematische *set*-Nachricht mit Inkrement und Behelfslösung

5.6 Erweiterung der Benutzeroberfläche

Für die Umsetzung einer symbolischen Ausführung wurde die Benutzeroberfläche von ScenarioTools angepasst. In Abbildung 5.6 ist die Erweiterung des Kontextmenüs zu sehen. Hier stehen nun zwei neue Algorithmen zur Zustandsraumerkundung zur Verfügung.

Abbildung 5.7 zeigt den *History View*, der symbolische Eigenschaften wie Zustands-Constraints anzeigen kann. Dabei wurde die Darstellung der

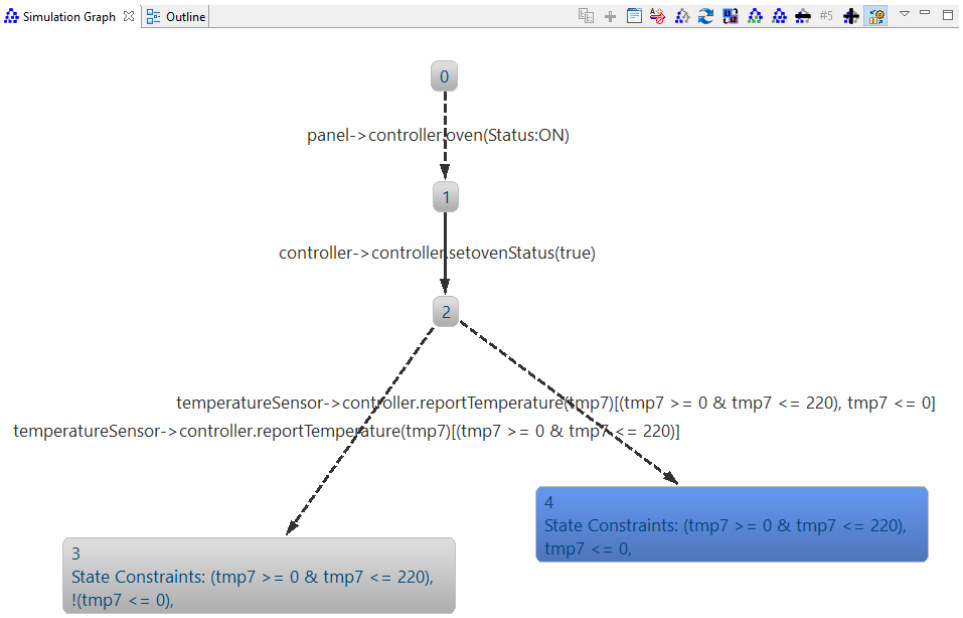

Abbildung 5.7: Erkundung eines Zustandsraums im *Play-Out-Modus* dargestellt im *HistoryView*

Constraints in eine besser lesbare Form transformiert als sie vom SMT-Solver zur Verfügung gestellt werden.

Die Erweiterung des *StateGraphViews* auf das symbolische Konzept ist in Abbildung 5.8 zu sehen. Symbolische Pfade zu Sicherheitsverletzungen sind hierbei in rot eingefärbt.

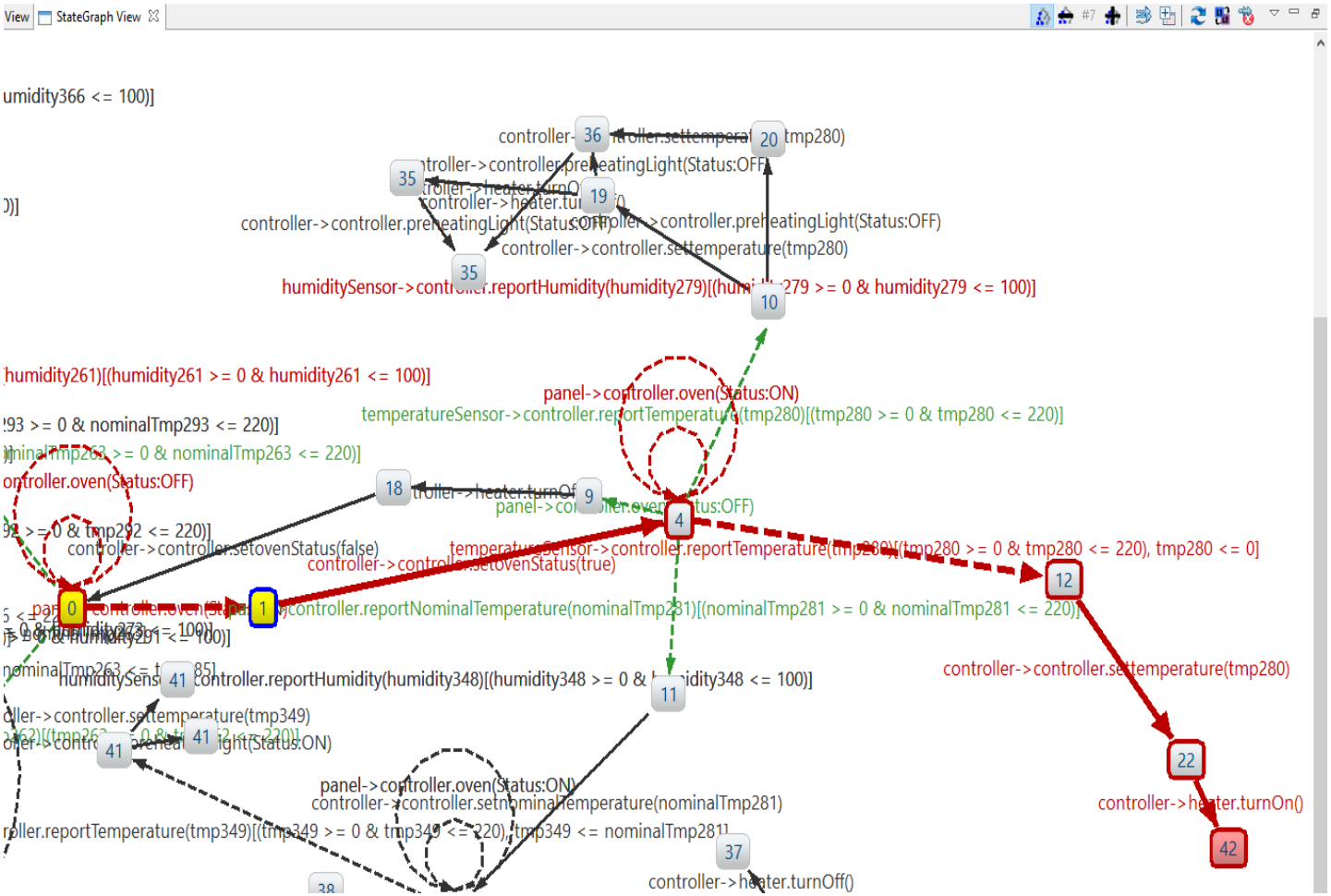

Abbildung 5.8: Darstellung eines erkundeten Zustandsraum im *Zustandsgraph-View*

Kapitel 6

Evaluation

In diesem Kapitel wird die symbolische Erweiterung von ScenarioTools an verschiedenen Beispielen mit Praxisbezug getestet. Dazu werden die Spezifikationen in verschiedenen Konfigurationen mehrfach hintereinander ausgeführt. Dabei werden die erkundeten Zustände sowie die zugehörigen Transitionen erfasst. Zudem wird die Zeit der Ausführungen gemessen und jeweils gemittelt. Die verschiedenen Konfigurationen zeichnen sich durch unterschiedlich große Wertebereiche von Attributen und Parametern aus. Der kleinste Wertebereich ist mit *W1* gekennzeichnet. Eine größere Nummer zeigt einen größeren Wertebereich an. Zudem gibt es bei allen Beispielen jeweils einen Vergleich der Performance zwischen symbolischer und expliziter Ausführung. An der *Anzahl der erkundeten Transitionen* kann die Anzahl der insgesamt erkundeten Zustände abgelesen werden. Diese erkundeten Zustände werden anschließend je nach Konfiguration mit gleichen oder ähnlichen Zuständen vereinigt. Die *Anzahl der Zustände* stellt somit die Anzahl der Zustände im resultierenden Zustandsgraphen dar. Der Wert der benötigten *Zeit pro Ausführung* wird aus 10 Ausführungen gemittelt. Alle Ausführungen, die nach 600 Sekunden nicht abgeschlossen sind, werden abgebrochen. Dadurch kann in diesen Fällen keine Anzahl der Transitionen und Zustände ermittelt werden. Dies wird in den Tabellen mit *n.b.* für *nicht bestimmt* dargestellt. Bei den angegebenen Mittelwerten ist zu berücksichtigen, dass sie nur als grobe Vergleichswerte herangezogen werden können. Dies wird auch von Păsăreanu et al. [42] betont. Auf Grund des *Garbage Collectors* von Java und vielen anderen nebenher laufenden Systemprozessen, seien die gemessenen Zeitwerte nicht sehr aussagekräftig, da sie stark variieren [42]. Păsăreanu et al. empfehlen daher eine Bewertung auf Grundlage der erkundeten Zustände. In dieser Arbeit erfolgte die Ausführung der Evaluation auf einem PC mit einem AMD Fx(tm)-6100 Six-Core Prozessor mit 3,3 GHz und 8GB Hauptspeicher. Als Betriebssystem wurde Windows 8.1 in der 64 Bit Version verwendet.

Die Evaluation setzt sich wie folgt zusammen. In Abschnitt 6.1 wird

eine Spezifikation einer Tunnelsteuerung getestet. Dieses Beispiel wurde für ScenarioTools ohne symbolische Ausführung entworfen. Für diese Arbeit wurde das Beispiel leicht modifiziert. Durch dieses Beispiel werden verschiedene Startkonfigurationen und variable Obergrenzen durch symbolische Objektattribute getestet. In Abschnitt 6.2 wird die Ofensteuerung evaluiert. Dabei werden als erstes verschieden große Parameterwertebereiche getestet. Als zweites wird ein Fehler in die Spezifikation der Ofensteuerung eingebaut. Hieran wird die Unterapproximation sowie die explizite Pfaderstellung aus symbolischen Zustandsgraphen getestet. Im dritten Abschnitt 6.3 wird ein Performancetest an einem relativ einfachen Beispiel durchgeführt. Dieses Beispiel wird in verschieden Varianten um unterschiedliche Bedingungen erweitert, um deren Auswirkungen auf die symbolische sowie die explizite Ausführung zu untersuchen. In Abschnitt 6.4 wird eine Spezifikation mit einer komplexen Formel geteste, die nicht-lineare Elemente enthält.

6.1 Tunnel Beispiel

Die Tunnel Spezifikation ist ein komplexes Beispiel aus dem Pool der Beispiel-Spezifikationen für ScenarioTools. Dabei geht es um eine Ampelsteuerung, die die Einfahrt in einen einspurigen Tunnel regelt. Zusätzlich zu den Ampellichtern verfügt das System über Sensoren, die in den Tunnel einfahrende und ausfahrende Autos melden und über Sensoren, die ankommende Autos erfassen. Die Aufgabe der Steuerung ist es, die Einfahrt der Autos so zu regeln, dass nie zwei Autos in entgegengesetzter Richtung gleichzeitig im Tunnel sind. In gleicher Richtung dürfen Autos nur bis zu einer maximalen Obergrenze in den Tunnel einfahren. Die Spezifikation sowie die Konfigurationsdatei ist im Anhang A.2 in den Listings A.3 und A.4 zu finden.

An diesem Beispiel soll die symbolische Initialisierung von Objektattributen evaluiert werden. Dazu wird die Anzahl der Autos, die sich aktuell im Tunnel befinden als symbolische Variable definiert. In der symbolischen Ausführung können dadurch zum einen alle möglichen Startbelegungen getestet werden. Zum anderen kann das Verhalten symbolisch getestet werden.

Für die Evaluation wird eine vollständige Zustandsraumerkundung ausgeführt. Diese wird sowohl symbolisch als auch explizit ausgeführt. Dabei wird die Obergrenze der Autos, die sich maximal im Tunnel befinden dürfen, mit jedem Durchlauf erhöht. Dieser Wert startet bei 1 und wird in den folgenden Schritten auf 2, 10 und 100 erhöht. Die symbolische Ausführung wird anschließend zusätzlich mit einer symbolischen Obergrenze ausgeführt ($Wmax$). Dadurch können alle möglichen Obergrenzen in einem Durchlauf getestet werden. Dies ist in der expliziten Ausführung nicht möglich.

In Tabelle 6.1 ist das Ergebnis der expliziten vollständigen Zustands-

Variante	Max #Autos	#Zustände	#Transitionen	Zeit(s)
W1 exp	1	74	264	1,709
W2 exp	2	102	362	2,107
W3 exp	10	326	1146	8,809
W4 exp	100	2846	9966	356,342

Tabelle 6.1: Ergebnisse der expliziten vollständigen Zustandsraumerkundung des Tunnel-Beispiels bei unterschiedlicher Anzahl erlaubter Autos im Tunnel

Variante	Max #Autos	#Zustände	#Transitionen	Zeit(s)
W1 sym	1	44	168	3,905
W2 sym	2	44	168	3,865
W3 sym	10	44	168	3,754
W4 sym	100	44	168	3,577
Wmax sym	100	44	168	3,664

Tabelle 6.2: Ergebnisse der symbolischen Zustandsraumerkundung des Tunnel-Beispiels bei unterschiedlicher Anzahl erlaubter Autos im Tunnel

raumerkundung dargestellt. Es ist zu sehen, dass der Zustandsraum mit zunehmender Obergrenze wächst. Ab Variante *W4* mit 100 Autos steigt die Explorationszeit zudem stark an. Zudem wurde der Wertebereich *Wmax* hier weggelassen. Dabei werden alle

Das Ergebnis der symbolischen Ausführung der vollständigen Zustandsraumerkundung ist in Tabelle 6.2 zu sehen. Wie erwartet hat der Wertebereich keinen Einfluss auf die Größe des Zustandsraums. Auch bei Hinzunahme der Obergrenze an Autos im Tunnel als symbolischen Wert in *Wmax sym* gibt es keine Veränderung der Größe des Zustandsraums. Beim Vergleich der benötigten Zeit ist zu erkennen, das bereits ab 10 Autos (*W3*) die symbolische Ausführung schneller ist.

Zudem werden in der symbolischen Ausführung alle möglichen Startkonfigurationen während einer Ausführung getestet. Um dies in der expliziten Ausführung durchzuführen, müsste sie für jede Startkonfiguration separat gestartet werden. Ein weiterer Vorteil des symbolischen Zustandsgraphen ist seine Größe. Diese lässt sich noch praktikabel vom Benutzer analysieren und überblicken.

In Abbildung 6.1 ist die für eine Zustandsraumexploration benötigte Zeit in Abhängigkeit von der Größe des Wertebereichs in einem Koordinatensystem aufgetragen. Dabei ist zu erkennen, dass sich die bei der symbolischen Ausführung benötigte Zeit konstant verhält. Die bei der expliziten Ausführung benötigte Zeit wächst hingegen drastisch.

Das Tunnel Beispiel zeigt, dass die symbolische Ausführung von Spezifikationen mit größeren Wertebereichen von Objektattributen sehr viel effektiver arbeitet als die explizite Ausführung. Dies ist zum einen an der

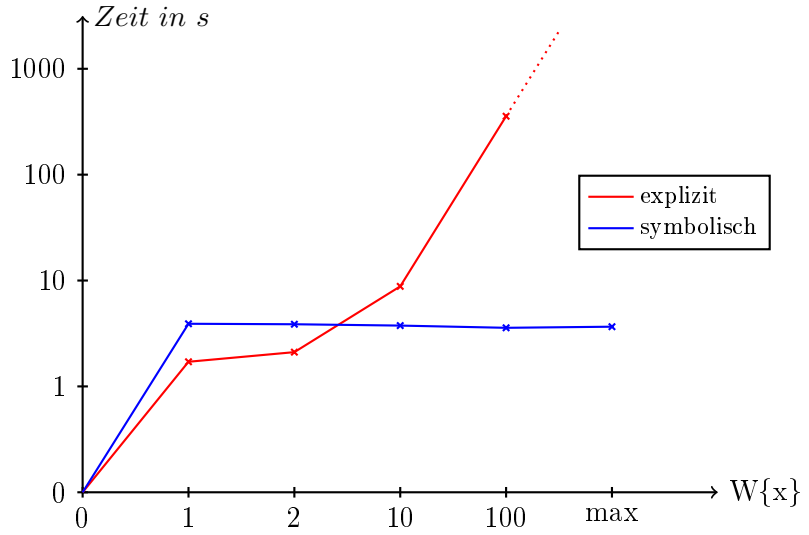

Abbildung 6.1: Benötigte Zeit für eine Zustandsraumerkundung bei größer werdenden Wertebereichen

geringeren Anzahl der erkundeten Zustände und an dem kleineren Zeitbedarf zu erkennen.

6.2 Ofen Beispiel

In diesem Abschnitt soll die Ofensteuerung evaluiert werden. Die vollständige Spezifikation ist im Anhang A.1 zu finden. In Abschnitt 2.3.2 wurden bereits die Anforderungen an die Spezifikationen erläutert. Die Komplexität dieses Beispieles entsteht durch mehrere verschiedene Parameterbereiche, die als Eingaben für die Ofensteuerung verstanden werden. Die Ofensteuerung reagiert auf gemessene Temperaturen und verfügt über einen Feuchtigkeitssensor, der die Luftfeuchtigkeit im Ofeninnenraum misst. Zudem kann die Solltemperatur des Ofens frei innerhalb des Wertebereiches gewählt werden. Für die Evaluation werden die Parameterwertebereiche wie in Listing 6.1 zu sehen eingeschränkt. Dabei wird der Wertebereich des Parameters *tmp* in den verschieden Durchläufen der Nachricht *reportTemperature()* angepasst. Im ersten Durchlauf umfasst der Wertebereich 0 bis 2. In den darauf folgenden Durchläufen wird er auf 0 bis 4, 8, 16, 32 und 64 erhöht. Der Wertebereich *Wmax* stellt den vollständigen Wertebereich dar. Dieser ist in der vollständigen Spezifikation im Anhang A.1 zu finden. Für die symbolische Ausführung werden alle Parameter der Nachrichten, die in Listing 6.1 zu sehen sind, als symbolische Parameter definiert.

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 exp	0 - 2	513	2496	39,189
W2 exp	0 - 4	960	6438	112,728
W3 exp	0 - 8	2208	23430	410,347
W4 exp	0 - 16	n.b.	n.b.	> 600
W5 exp	0 - 32	n.b.	n.b.	> 600
W6 exp	0 - 64	n.b.	n.b.	> 600
Wmax exp	0 - 220	n.b.	n.b.	> 600

Tabelle 6.3: Ergebnisse der expliziten vollständigen Zustandsraumerkundung der Ofensteuerung

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 sym	0 - 2	209	624	40,878
W2 sym	0 - 4	161	471	25,284
W3 sym	0 - 8	161	471	26,043
W4 sym	0 - 16	161	471	25,517
W5 sym	0 - 32	161	471	25,262
W6 sym	0 - 64	164	478	25,703
Wmax sym	0 - 220	233	678	57,878

Tabelle 6.4: Ergebnisse der symbolischen Zustandsraumerkundung der Ofensteuerung

```
1 parameter ranges {
2   Controller.reportTemperature(tmp = [ 0 .. 16]),
3   Controller.reportNominalTemperature(nominalTmp = [ 0 .. 2 ]),
4   Controller.reportHumidity(humidity = [ 0 .. 1 ])
5 }
```

Listing 6.1: Parameterwertebereiche der Ofensteuerung für den Wertebereich *W4*

In Tabelle 6.3 ist das Ergebnis der expliziten vollständigen Zustandsraumerkundung zu sehen. Der Zustandsraum wächst, wie erwartet, mit der Größe des Parameterwertebereichs. Ab Wertebereich *W4* kann die Zustandsraumerkundung innerhalb des Zeitlimits kein Ergebnis mehr liefern. Beim Betrachten der erkundeten Zustände im Wertebereich *W3* ist zu erkennen, dass der Zustandsraum sehr groß geworden ist.

Das Ergebnis der symbolischen Zustandsraumerkundung ist in Tabelle 6.4 zu sehen. Auch hier ist wieder zu beobachten, dass der veränderte Wertebereich fast keinen Einfluss auf den erkundeten Zustandsgraphen hat. Bei Wertebereich *W1* fällt ein erhöhter Wert der erkundeten Zustände auf. Dies hängt mit den Wertebereichen von Temperatur und Solltemperatur zusammen. In diesem Fall sind sie genau gleich. Dies wirkt sich auf die Interpretation der Spezifikation aus. Ebenso weist der Wertebereich *Wmax*

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 exp sv	0 - 2	498	2539	49,111
W2 exp sv	0 - 4	994	6577	114,191
W3 exp sv	0 - 8	2201	23821	410,532
W4 exp sv	0 - 16	n.b.	n.b.	> 600
W5 exp sv	0 - 32	n.b.	n.b.	> 600
W6 exp sv	0 - 64	n.b.	n.b.	> 600
Wmax exp sv	0 - 220	n.b.	n.b.	> 600

Tabelle 6.5: Ergebnisse der expliziten Zustandsraumerkundung des Ofen-Beispiels mit Sicherheitsverletzung

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 sym sv	0 - 2	184	559	33,223
W2 sym sv	0 - 4	172	559	29,603
W3 sym sv	0 - 8	172	559	29,723
W4 sym sv	0 - 16	172	559	29,686
W5 sym sv	0 - 32	172	559	29,838
W6 sym sv	0 - 64	175	566	30,177
Wmax sym sv	0 - 220	200	631	35,167

Tabelle 6.6: Ergebnisse der symbolischen Zustandsraumerkundung des Ofen-Beispiels mit Sicherheitsverletzung

einen erhöhten Wert der erkundeten Zustände auf. Dieses Mal kommt hinzu, dass einige Aktionen von Szenarien, auf Grund der erhöhten Wertebereiche, ausgeführt werden können. Bei kleineren Wertebereichen war dies zuvor nicht möglich. Das Szenario *HumidityRegulation* wird beispielsweise nach Aktivierung nur weiter ausgeführt, wenn die gemessene Luftfeuchtigkeit immer noch größer als 50% ist.

Im Folgenden wird nun eine Sicherheitsverletzung in die Ofen-Spezifikation eingebaut. Dazu werden die Regeln für die Deaktivierung der Vorheizlampe so verändert, dass eine Definitionslücke entsteht. Siehe dazu Szenario *PreheatLightOff*. Im korrekten Fall wird das Szenario nicht weiter ausgeführt, wenn die gemessene Temperatur kleiner oder gleich der Solltemperatur ist. Für die Sicherheitsverletzung wird der Fall, dass die Werte gleich sind, entfernt. Tritt nun der Fall auf, dass die beiden Werte gleich sind, werden zwei Nachrichten vom System gefordert die unterschiedliche Parameter haben. Dies führt dazu, dass diese Nachrichten nicht ausgeführt werden können. Wenn keine anderen Systemnachrichten ausgeführt werden können, bedeutet dies eine Sicherheitsverletzung.

Die Ergebnisse der expliziten Zustandsraumerkundung der Ofen Spezifikation mit Sicherheitsverletzung sind in Tabelle 6.5 zu sehen. Diese ähneln denen mit fehlerfreier Spezifikation. Ein ähnliches Verhältnis ist bei den

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 sym sv	0 - 2	138	446	21,086
W2 sym sv	0 - 4	138	446	21,680
W3 sym sv	0 - 8	138	446	21,193
W4 sym sv	0 - 16	138	446	21,166
W5 sym sv	0 - 32	138	446	21,050
W6 sym sv	0 - 64	141	452	21,760
Wmax sym sv	0 - 220	148	466	22,785

Tabelle 6.7: Ergebnisse der symbolischen Unterapproximation des Ofen-Beispiels mit Sicherheitsverletzung

Variante	Wertebereich	#Zustände	#Transitionen	Zeit(s)
W1 sym sv	0 - 2	62	108	6,653
W2 sym sv	0 - 4	64	110	6,904
W3 sym sv	0 - 8	64	110	6,864
W4 sym sv	0 - 16	64	110	6,857
W5 sym sv	0 - 32	64	110	6,635
W6 sym sv	0 - 64	54	95	6,595
Wmax sym sv	0 - 220	55	96	6,777

Tabelle 6.8: Ergebnisse der Erstellung von konkreten Pfaden zu einer Sicherheitsverletzung während einer symbolischen Zustandsraumerkundung

Ergebnissen der symbolischen Zustandsraumerkundung in Tabelle 6.6 zu erkennen. Die Sicherheitsverletzungen werden in beiden Fällen zuverlässig entdeckt. Ausgenommen sind die Durchläufe, die nicht in der vorgesehenden Zeit geblieben sind.

Für die Ofensteuerung mit Sicherheitsverletzung bietet es sich an, einen Schnelltest durchzuführen. Dazu wird die Spezifikation symbolisch mit einer Unterapproximation getestet. In Tabelle 6.7 ist das Ergebnis der Unterapproximation zu sehen. Dabei wurden ungefähr ein Fünftel weniger Zustände erkundet. Dieser Unterschied lässt sich auch in der benötigten Zeit ablesen, denn es wurde nur ungefähr zwei Drittel der Zeit benötigt. Dies ist auch darin begründet, dass der SMT-Solver bei diesem Verfahren während der Zustandsverschmelzung nicht benötigt wird. Die Unterapproximation hat in allen Durchläufen erfolgreich die Sicherheitsverletzung entdeckt.

Bei dieser Spezifikation kann auf Grund der vorhandenen Sicherheitsverletzung ein konkreter Pfad aus einem symbolischen Zustandsgraphen erstellt werden. Für die Exploration des Zustandsgraphen wurde eine iterative Tiefensuche eingesetzt, die beim Erkunden einer Sicherheitsverletzung abbricht und den explorierten Teilgraphen für die Erstellung des Pfades nutzt.

Die Ergebnisse der Pfaderstellung aus der Ofen-Spezifikation sind in Tabelle 6.8 zu sehen. Auch hier fallt auf, dass bei verschiedenen Varianten

nahezu die Gleiche Anzahl an Zuständen erkundet wurde. Die Anzahl der erkundeten Zustände lässt sich für jedes Beispiel weiter optimieren, indem die Suchtiefe pro Iteration der iterativen Tiefensuche angepasst wird. Für die Evaluation wurde der voreingestellte Wert 10 benutzt. In Listing 6.2 ist der synthetisierte Pfad zu sehen. Er führt zur oben beschriebenen Sicherheitsverletzung. Allerdings ist dies nicht der kürzeste Pfad. Für den kürzesten Pfad muss die Suchtiefe auf 1 gesetzt werden. Dadurch übernimmt die iterative Tiefensuche die Eigenschaften einer Breitensuche. Dann werden in diesem Beispiel jedoch ungefähr doppelt so viele Zustände erkundet.

```
1  Init Values
2  Path to Violation
3  panel->controller.oven(Status:ON)
4  controller->controller.setovenStatus(true)
5  temperatureSensor->controller.reportTemperature(1)
6  controller->controller.settemperature(1)
7  controller->heater.turnOff()
8  controller->controller.preheatingLight(Status:OFF)
9  humiditySensor->controller.reportHumidity(0)
10 temperatureSensor->controller.reportTemperature(0)
11 controller->controller.settemperature(0)
12 controller->heater.turnOn()
```

Listing 6.2: Konkreter Pfad zu einer Sicherheitsverletzung erstellt aus der symbolischen Analyse

Das Beispiel der Ofensteuerung zeigt, dass die symbolische Ausführung von Spezifikationen mit größeren Wertebereichen von Parametern sehr viel effektiver arbeitet als die explizite Ausführung. Dies ist daran zu erkennen, dass die symbolische Ausführung in der Lage ist gewisse Konfigurationen (*W4* - *Wmax*) vollständig in der gegebenen Zeit zu erkunden. Zudem werden die einzelnen Schritte des iterativen Modellierungsprozesses. Fehler können in der symbolischen Ausführung erkannt und auf eine konkrete Art mitgeteilt werden.

6.3 Kaskadierendes Beispiel

In diesem Beispiel soll die Performance der symbolischen Ausführung direkt mit der Performance der expliziten Ausführung verglichen werden. Zudem sollen Faktoren getestet werden, die eine symbolische Ausführung von Spezifikationen erschweren und es soll getestet werden, welche Faktoren sich am stärksten auf die explizite Ausführung auswirken. Dazu wird ein einfaches generisches Beispiel benutzt, welches um Eigenschaften erweitert wird, die in Bezug auf die Performance von expliziten und symbolischen Ausführungen untersucht werden soll. Die Werte werden mit der vollständigen Zustandsraumexploration erfasst.

Das Listing 6.3 zeigt den Aufbau des Beispiels sowie die verschiedenen Varianten. Ein Beispiel einer vollständigen Kaskade von *Variante A* ist im Anhang A.3 zu sehen. Das Beispiel besteht aus einem Starterszenario

Cascade1 und vier weiteren Szenarien einer bestimmten Variante. Dabei bekommen die Nachrichten in einer höheren Ebene jeweils eine höhere Nummer. Das Starterszenario wird durch eine Umweltnachricht mit einem Parameter gestartet. Diesem Parameter wird ein Parameterwertebereich zugewiesen. Anschließend starten die folgenden beiden Nachrichten jeweils das Szenario *Cascade2*. Ist das zweite Szenario gestartet, werden mit den zwei folgenden Nachrichten ebenfalls zwei Szenarien der nächst höheren Ebene gestartet. Dies setzt sich bis zur letzten Kaskade fort. Dabei muss das Szenario der nächst höheren Ebene allerdings erst wieder beendet sein bevor es ein zweites Mal gestartet werden kann. Dies liegt daran, dass die Startnachricht solange geblockt ist, wie ein Szenario der nächst höheren Ebene aktiv ist. Für die Evaluation wird eine vierstufige Kaskade genutzt. Für die einzelnen Durchläufe wird der Parameterwertebereich von dem Einzelwert 25 auf 0 bis 10, 100 und 100.000 erhöht. In weiteren Durchläufen werden die Szenarien *Cascade2* und höher mit folgenden Bedingungen erweitert. Dabei ist *Variante A* die Ausgangsvariante. *Variante B* ist um eine Bedingung erweitert, die das Szenario abbricht, wenn der Wert kleiner als 20 ist. In *Variante C* kommt eine zusätzliche Bedingung hinzu, die das Szenario beendet wenn, der Wert größer als 90 ist. Die letzte Variante (*Variante D*) lockert die Ausführungsreihenfolge der Nachrichten auf. Durch das Szenario-Constraint können Szenarien der gleichen Kaskadenstufe gleichzeitig aktiv sein, da die Nachricht, die es aktiviert, nicht mehr von anderen Szenarien geblockt werden kann.

In Tabelle 6.9 ist das Ergebnis der **Variante A** zu sehen. Als erstes fällt auf, dass die Anzahl der erkundeten Transitionen und die Anzahl der Zustände im Wertebereich *W1* sowohl bei der expliziten als auch bei der symbolischen Ausführung identisch sind. Bei der expliziten Ausführung steigen die Werte für erkundete Transitionen und Zustände mit größer werdendem Wertebereich, bis es beim Wertebereich *W4* zum Timeout kommt. Die Werte der symbolischen Ausführung bleiben hingegen konstant.

Das Ergebnis der Durchführung von **Variante B** ist in Tabelle 6.9 zu sehen. Die Anzahl der explizit erkundeten Transitionen sowie die der Zustände sind im Vergleich zu *Variante A* deutlich geringer. In der symbolischen Ausführung kommt, im Vergleich zu *Variante A*, lediglich eine Transition hinzu.

Die Tabelle 6.9 zeigt das Ergebnis der Durchführung von **Variante C**. Die erkundeten Zustände gehen beim Wertebereich *W3* im Vergleich zu *Variante B* in der expliziten Durchführung um gute 10% zurück. Bei der symbolischen Ausführung hingegen verdoppelt sich die Anzahl der Zustände. Dies liegt an der eingefügten Bedingung, da diese einen Split zur Folge hat. Der resultierende Zustand kann nicht mit einem bereits vorhandenen Zustand vereinigt werden.

Das Ergebnis der Zustandsraumerkundung von **Variante D** ist in Tabelle 6.9 zu sehen. Die Anzahl der erkundeten Zustände sowie die der

Transitionen verändert sich immer dann, wenn der Wertebereich vergrößert wird und somit Bedingungen die Splitbedingung erfüllen.

Der Parameterbereich ist sehr entscheidend für die konkrete Ausführung. Je größer der Parameterbereich ist, umso größer ist der Zustandsraum. Für die symbolische Ausführung ist der Parameterbereich weniger relevant. Er hat nur dann Einfluss auf die Größe des Zustandsraums, wenn dadurch Bedingungen erfüllbar werden. Damit kann auch mit diesem generischen Beispiel gezeigt werden, dass die symbolische Ausführung Vorteile gegenüber der expliziten Ausführung zu bieten hat, wenn es um Spezifikationen mit großen Parameterwertebereichen geht.

```
// Anzahl an Cascaden 4
parameter ranges {
  A.startCascade(value = [ 25 ]) // 0..50, 0..100, 0..100 000
}
...
// Startkaskade
specification scenario Cascade1 {
  var EInt value
  message e->a.startCascade(bind to value)
  message strict requested a->b.msg1(value)
  message strict requested a->b.msg1(value)
}

// Variante A
specification scenario Cascade2 {
  var EInt value
  message a->b.msg1(bind to value)
  message strict requested a->b.msg2(value + 1)
  message strict requested a->b.msg2(value + 1)
}

// Variante B
specification scenario Cascade2 {
  var EInt value
  message a->b.msg1(bind to value)
  interrupt if [value < 20]
  message strict requested a->b.msg2(value + 1)
  message strict requested a->b.msg2(value + 1)
}

// Variante C
specification scenario Cascade2 {
  var EInt value
  message a->b.msg1(bind to value)
  interrupt if [value < 20]
  message strict requested a->b.msg2(value + 1)
  interrupt if [value > 90]
  message strict requested a->b.msg2(value + 1)
}

// Variante D
specification scenario Cascade2 {
  var EInt value
  message a->b.msg1(bind to value)
  interrupt if [value < 20]
  message strict requested a->b.msg2(value + 1)
  interrupt if [value > 90]
  message strict requested a->b.msg2(value + 1)
}constraints [ ignore message a->b.msg1(*)]
```

Listing 6.3: Einzelne Szenarien der verschiedenen Varianten der Kaskade

Variante	**Parameterbereich**	**#Zustände**	**#Transitionen**	**Zeit(s)**
W1 exp	25	45	59	3,502
W2 exp	0 - 50	2245	3009	6,223
W3 exp	0 - 100	4445	5959	9,566
W4 exp	0 - 100000	n.b.	n.b.	> 600
W1 sym	25	45	59	7,577
W2 sym	0 - 50	45	59	7,639
W3 sym	0 - 100	45	59	7,156
W4 sym	0 - 100000	45	59	7,948

Tabelle 6.9: Ergebnisse der Kaskade in Variante A

Variante	**Parameterbereich**	**#Zustände**	**#Transitionen**	**Zeit(s)**
W1 exp	25	45	59	0,177
W2 exp	0 - 50	1365	1849	3,111
W3 exp	0 - 100	3565	4799	7,423
W4 exp	0 - 100000	n.b.	n.b.	> 600
W1 sym	25	45	59	9,034
W2 sym	0 - 50	45	60	9,149
W3 sym	0 - 100	45	60	9,047
W4 sym	0 - 100000	45	60	9,505

Tabelle 6.10: Ergebnisse der Kaskade in Variante B

Variante	**Parameterbereich**	**#Zustände**	**#Transitionen**	**Zeit(s)**
W1 exp	25	45	59	0,785
W2 exp	0 - 50	1365	1849	2,979
W3 exp	0 - 100	3127	4207	6,558
W4 exp	0 - 100000	n.b.	n.b.	> 600
W1 sym	25	45	59	10,771
W2 sym	0 - 50	45	60	11,530
W3 sym	0 - 100	91	123	21,275
W4 sym	0 - 100000	91	123	20,297

Tabelle 6.11: Ergebnisse der Kaskade in Variante C

Variante	Parameterbereich	#Zustände	#Transitionen	Zeit(s)
W1 exp	25	80	184	2,402
W2 exp	0 - 50	2490	5764	14,052
W3 exp	0 - 100	5668	13074	24,650
W4 exp	0 - 100000	n.b.	n.b.	> 600
W1 sym	25	80	184	43,086
W2 sym	0 - 50	81	186	78,765
W3 sym	0 - 100	117	279	91,475
W4 sym	0 - 100000	117	279	90,126

Tabelle 6.12: Ergebnisse der Kaskade in Variante D

6.4 Nicht-lineare Integer-Arithmetik

Dieses Beispiel zeigt die Möglichkeit der symbolischen Ausführung von Spezifikationen mit schwierigen Ausdrücken. SMT-Solver sind mittlerweile in der Lage sehr große und schwierige Gleichungen zu lösen [28]. Deswegen soll dieses Beispiel nur einen kleinen Einblick dahin geben, was möglich ist. Das Beispiel in Listing 6.4 zeigt ein Szenario, das anhand einer Reihe gemessener Temperaturen und unter Berücksichtigung der Zeit eine neue Solltemperatur errechnet. Dies geschieht auf Grundlage einer fiktiven Temperaturkurve.

```
1  parameter ranges {
2    Controller.sensorValues(tmp = [0..220], tmpIn = [0..220], time = [0..300])
3  }
4  ...
5  specification scenario temperatureCurveForTurkey {
6    var EInt tmp
7    var EInt tmpIn
8    var EInt time
9    message ts -> ctr.sensorValues(bind to tmp, bind to tmpIn, bind to time)
10
11   var EInt newTmp =   10*tmp*tmp*tmp*tmp*tmp
12                     -  4*tmpIn*tmpIn*tmpIn*tmpIn
13                     - 12*time*time*time*time
14
15   interrupt if [newTmp > tmp + 10]
16   interrupt if [newTmp > 220]
17   message strict requested ctr -> ctr.reportNominalTemperature(newTmp)
18 }
```

Listing 6.4: Szenario mit nicht-linearer Arithmetik

Bei der symbolischen Zustandsraumerkundung der Spezifikation werden zwei Zustände und drei Transitionen erkundet. Die gesamte Erkundung benötigt 14,128 Sekunden. Bei der expliziten Ausführung lässt sich die Simulation nicht starten. Dies liegt an der großen Menge möglicher Umweltnachrichten, die durch die Parameterwertebereiche definiert werden. Dies sind insgesamt 14.701.141 Nachrichten. Wird die initialisierende Nachricht auf drei Nachrichten mit jeweils einem Parameter aufgeteilt, dann kann auf diese Weise das Problem umgangen werden. Anschließend werden zusätzlich die Parameterwertebereiche auf $tmp = [0..220]$, $tmpIn = [11]$, $time =$

[12] eingeschränkt. Dadurch lässt sich die Simulation in angemessener Zeit ausführen. Jedoch ist die explizite Ausführungslogik nicht in der Lage diese Gleichungen korrekt zu lösen. Sie ist darauf angewiesen mit konkrete Werten zu rechnen. In diesem Fall muss der Variable *newTmp* ein konkreten Wert zugewiesen werden muss. Dies führt zu einem Integerüberlauf und somit zu falschen Ergebnissen. Die symbolische Ausführungslogik hingegen ist nicht darauf angewiesen konkrete Werte auszurechnen. Dadurch kommt es hier nicht zu einem Integerüberlauf. Damit ist die symbolische Ausführung in der Lage diese Spezifikation korrekt zu explorieren, beachtet jedoch nicht die Möglichkeit eines Integerüberlaufs. Da ein echtes System an dieser Stelle den errechneten Wert zwischenspeichern muss, wäre es hier von Vorteil, wenn die Möglichkeit eines Integerüberlaufs erkannt wird. Die symbolische Ausführung kann für dieses Beispiel eine Behelfslösung nutzten. Dafür wird nach der Berechnung des Wertes für *newTmp* folgende Überprüfung eingefügt:

```
1 violation if [newTmp > 2147483647] // newTmp > Integer.MAX => Integerüberlauf
```

Dadurch wird eine Sicherheitsverletzung ausgelöst, wenn die Möglichkeit eines Integerüberlaufs besteht. In zukünftigen Versionen von ScenarioTools sollte dies allerdings intern gelöst werden. Damit kann jedoch gezeigt werden, dass die symbolische Ausführung durch die Behelfslösung in der Lage ist dieses Problem zu erkennen.

Kapitel 7

Verwandte Arbeiten

In diesem Kapitel werden zu dieser Arbeit verwandte Arbeiten beschrieben. Zum einen gibt es ähnliche Arbeiten aus dem Bereich der szenariobasierten Modellierung und dem Prüfen von Modellen. Zum anderen gibt es Arbeiten, die sich mit der symbolischen Ausführung beschäftigen.

Von Zurowska und Dingel werden in [50] UML-RT Zustandsmaschinen symbolisch ausgeführt. Zudem kann während der Ausführung von Transitionen *Action Code* ausgeführt werden. Dieser wird als Funktion abstrahiert und kann auf diese Weise ebenso symbolisch ausgeführt werden. Die symbolische Analyse wird dabei von KLEE [10] übernommen. Zum Lösen der symbolischen Gleichungen wird der Choco Solver [39] verwendet. Im Unterschied zu dieser Arbeit wird bei Zurowska und Dingel [50] zur Zustandsvereinigung lediglich ein Zeichenkettenvergleich der Zustands-Constraints durchgeführt. Eine Teilmengenprüfung findet nicht statt.

Rapin beschreibt in [43] fundamentale Elemente der symbolische Ausführung von *Input Output Symbolic Transition Systems*. Diese werden mithilfe von Formeln der temporalen Logik verifiziert.

In Marelly et al. [36] werden symbolische Variablen in LSCs beschrieben. Dabei handelt es sich allerdings nicht um eine symbolische Ausführung im eigentlichen Sinne. Vielmehr ist das beschriebene Konzept mit der konkreten Variante der *Bind-to-Parameter* in ScenarioTools zu vergleichen. Dabei können die Variablen als Parameter in weiteren Nachrichten des Szenarios wiederverwendet werden.

Von Wang et al. [49] wird ein Konzept zur symbolischen Ausführung von LSC vorgestellt. Dabei werden Variablen, die Daten beschrieben und Variablen, die Zur Kontrolle dienen symbolisch betrachtet. Auch diskrete Zeit kann mit dem Konzept symbolisch betrachtet werden.

Harel et al. [25] beschäftigen sich mit der symbolische Ausführung von Spezifikationen, die mit dem *Behavourial Programming* (BP) Framework erstellt wurden. Bei BP handelt es sich um eine szenariobasierte Entwicklungsmethode, wobei Szenarien in einer Programmiersprache (wie beispielsweise

JAVA oder C) umgesetzt werden. Das beschriebene Konzept sieht vor, dass Szenarien (hier *b-threads* genannt) durch ein Z3 Modell definiert werden. Dadurch kann jeder *b-thread* separat getestet werden, indem er gegen das Z3 Model getestet wird. Dann kann Z3 das Zusammenspiel aller b-threads auf Grundlage der einzelnen Z3 Modelle testen. Dies ist effektiver als eine Zustandsraumerkundung. Wenn ein *b-thread* erweitert werden soll, kann das Z3 Modell wie bei der Test getriebenen Entwicklung erweitert werden. Das Z3 Modell stellt dabei den Test dar. Solange der zugehörige *b-thread* den Test nicht erfüllt, wird ein Beispiel-Pfad geliefert, der den Fehler beschreibt.

Katz [29] stellt ein Verfahren vor, dass den Zustandsraum einer BP Spezifikation mithilfe einer Überapproximation einschränkt. Die Überapproximation wird anschließend durch ein weiteres Verfahren *repariert*. Dadurch werden unechte Pfade und Pfade, die zu Fehlern führen, entfernt. Im Gegensatz zu dieser Arbeit werden Nachrichtenparameter nicht betrachtet.

Der *Java PathFinder* (JPF) [30, 48] ist ein Tool, das für die Zustandsraumerkundung von JAVA Programmen entwickelt wurde. Mit der Erweiterung *Symbolic PathFinder* [41, 42] wird das Tool in die Lage versetzt, JAVA Programme symbolisch auszuführen und zu analysieren. Es nutzt dabei Techniken zum Umgang mit komplexen Eingabedatenstrukturen, Zeichenketten und native Aufrufe zu externen Bibliotheken. Zudem kann JPF *Standard UML* und *Rhapsody UML* symbolisch analysieren, indem die Modelle zu Java transformiert werden.

KLEE [10] ist ein weiteres Tool zur symbolischen Ausführung. Bei der Kompilierung muss ein bestimmter Zwischencode (LLVM [34]) erzeugt werden, der von KLEE symbolisch analysiert werden kann. In diesem Zwischencode kann nach unerlaubten Anweisungen, wie das Dividieren durch Null oder Speicherzugriff auf nicht reservierten Speicher, gesucht werden. Das auf KLEE basierende symbolisches Ausführungsprogramm EXE [19, 11] ist für das Erstellen von konkreten Pfaden zu Fehlern bekannt geworden. Cloud9 [33, 12] ist ein weiteres KLEE basierendes Tool, das zum Testen von großer Software ausgelegt ist. Dafür wird die symbolische Ausführung parallelisiert, wodurch sie auf einem Computercluster skalierbar wird.

Zudem gibt es eine ganze Reihe an symbolischen Ausführungsprogrammen, die den Bytecode von Programmen analysieren können. Dazu zählt unter anderem MergePoint [3]. Es arbeitet auf 32-bit Linux Binärdateien und benötigt keine speziellen Informationen über das zu analysierende Programm. DART [20] (*Directed Automated Random Testing*) ist ein Programm zur Testfallerstellung. Durch CUTE [45] und jCUTE (CUTE für JAVA) [44] wurde der Begriff *Concolic Execution* geprägt. SAGE [17] ist ein symbolisches Testprogramm von Microsoft. Es hat ein Drittel aller Fehler während der Entwicklung von Windows 7 entdeckt. Für .Net Programme gib es das symbolische Ausführungsprogramm PEX [47].

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

ScenarioTools ist ein Programm zur Erstellung und Analyse von szenariobasierten Spezifikationen für verteilte reaktive Systemen. Dabei werden die spezifizierten Systeme mit der Modellierungssprache *Scenario Modeling Language* (SML) beschrieben. Viele der spezifizierten Systeme erhalten parametrisierte Nachrichten mit großen Integer-Parameterwertebereichen aus ihrer Umwelt. Die Menge dieser Spezifikationen kann aktuell nicht effizient analysiert werden, da sie unter der Zustandsraumexplosion leidet.

Dieses Problem wurde in der vorliegenden Arbeit in Angriff genommen. Dazu wurde ein Konzept zur symbolischen Ausführung von SML-Spezifikationen entwickelt. Dabei können Nachrichtenparameter und Objektattribute vom Typ Integer symbolisch betrachtet werden. Dadurch können große Mengen an parametrisierten Nachrichten zusammengefasst werden. Auf diese Weise werden Gruppen von Zuständen in einem Schritt erkundet und zusammengefasst, sodass einer Zustandsraumexplosion vorgebeugt wird. Zudem können mit der symbolischen Betrachtung von Objektattributen verschiedene Startkonfigurationen vom Objektsystem mit einer Ausführung analysiert werden. Außerdem werden bei der symbolischen Analyse Pfade zu Fehlern erstellt, die anschließend in Pfade mit konkreten Werten umgewandelt werden können.

Dieses Konzept wurde in einem Prototypen umgesetzt und in ScenarioTools integriert. Dabei wurde für das Lösen der komplexen Formeln, die während der symbolischen Ausführung entstehen, der Z3 SMT-Solver verwendet. Der Prototyp implementiert verschiedene Verfahren zur Vereinigung von Zuständen. Dazu zählen eine Zustandsvereinigung mit Teilmengenvergleich. Dabei werden symbolische Zustände zusammengefasst, die die gleiche Menge an konkreten Zuständen repräsentieren oder Zustände bei denen

der eine Zustand eine Teilmenge des anderen darstellt. Hierbei findet eine vollständige Erkundung des Zustandsraums statt. Des Weiteren gibt es eine Zustandsvereinigung mittels Unterapproximation, bei der ähnliche Zustände zusammengefasst werden. Dadurch ist die Menge der erkundeten Zustände nicht mehr vollständig. Der Vorteil dieses Verfahrens liegt jedoch darin, dass es weniger rechenintensiv als das vorherige Verfahren ist und trotzdem ein hohes Potential, aufweist eine Sicherheitsverletzung zu finden. Zudem kann die symbolische Zustandsvereinigung deaktiviert werden. Dies bringt Vorteile wenn Formeln, die während der symbolischen Ausführung entstehen, nicht mehr effizient gelöst werden können. Auf diese Weise kann trotzdem eine Analyse stattfinden.

Zur Zustandsraumerkundung wurden zwei Verfahren implementiert: Ein Verfahren zur vollständigen Zustandsraumerkundung und ein Verfahren, das die iterative Tiefensuche verwendet und beim Erkunden einer Sicherheitsverletzung abbricht. Dies lässt sich gut mit der Einstellung der deaktivierten symbolischen Zustandsvereinigung kombinieren. Das Konzept wurde so integriert, dass ein Wechsel zwischen symbolischer und expliziter Ausführung durch eine einfache Erweiterung der Konfigurationsdatei möglich ist. Dabei können die Parameter und Objektattribute, die symbolisch betrachtet werden sollen, in der Konfigurationsdatei eingefügt werden. Die Spezifikation muss dabei nicht angepasst werden. Zudem kann die Art der Zustandsvereinigung in der Konfigurationsdatei gewählt werden.

Die Implementierung wurde anhand von Beispielen evaluiert und mit der expliziten Ausführung verglichen. Bei der Evaluation wurden verschiedene Spezifikationen ausgeführt und dabei die Größe des Zustandsraums sowie die benötigte Zeit verglichen. Dabei wurde eine Spezifikation evaluiert, bei der der Schwerpunkt auf symbolischen Objektattributen lag, und eine Spezifikation untersucht, die viele verschiedene parametrisierte Nachrichten aus der Umwelt erhält. Darüber hinaus wurde ein generisches Beispiel betrachtet, das in verschiedenen Varianten erweitert wurde. Dadurch konnten Aussagen über die Auswirkung einzelner Anweisungen auf die explizite und symbolische Ausführung gemacht werden. Die gesamte Evaluation kam zu dem Ergebnis, dass die erkundeten Zustandsräume bei der symbolischen Ausführung wie gewünscht deutlich geringer als bei der expliziten waren. Darüber hinaus zeigte die symbolische Ausführung ab einer gewissen Größe der Parameterwertebereiche bemerkenswerte Performancevorteile gegenüber der expliziten Ausführung. Dabei erkundete sie Zustandsräume innerhalb von Sekunden, die von der expliziten Ausführung in zehn Minuten nicht erkundet werden konnten.

Zusammenfassend wurde durch dieser Arbeit ScenarioTools in die Lage versetzt, Spezifikationen effizient zu analysieren, die parametrisierte Eingaben aus der Umwelt mit großen Integer-Parameterwertebereichen aufweisen.

8.2 Ausblick

In Zukunft kann die symbolische Ausführung in ScenarioTools mit folgenden zusätzlichen Funktionen erweitert werden:

Es ist bereits möglich aus symbolisch erkundeten Pfaden konkrete Pfade zu erzeugen. Dieses Verfahren könnte so erweitert werden, dass Testfälle bzw. -pfade bis zu einer gewissen Länge erstellt werden, um damit spätere Implementierungen testen zu können. Von Panzica et al. [35] wurde bereits ein Verfahren vorgestellt, bei dem bestimmte Pfade eines Controllers ausgewählt werden, mit denen eine gute Testabdeckung erreicht werden kann. Durch dieses Verfahren könnten Testpfade gezielt ausgewählt werden, um die beschrieben Testabdeckung zu erfüllen.

ScenarioTools unterstützt aktuell nur die Verarbeitung von ganzen Zahlen, da dies in der konkreten Ausführung besser handhabbar ist als Fließkommazahlen. Beispielsweise ist es praktisch nicht sinnvoll alle Nachrichten mit einem Parameterwertebereich von 0 bis 1 mit Fließkommazahlen aufzuzählen. Die SMT-Solver die während der symbolischen Ausführung genutzt werden können besser mit Fließkommazahlen rechnen. Daher wäre es in Zukunft erstrebenswert Fließkommazahlen in das Repertoire von ScenarioTools mit aufzunehmen.

Aktuell gibt es zwei Erweiterungen von ScenarioTools, die bei der Ausführung von Nachrichten Aktionen ausführen können. Diese werden momentan nicht von der symbolischen Ausführung unterstützt. Die erste Erweiterung nutzt das Transformationstool *Henschin* um Veränderungen des Objektsystems durchzuführen [22]. Die zweite Erweiterung erlaubt dem Benutzer die Ausführung von Javacode. Da diese Erweiterungen keine symbolischen Werte als Eingabe verarbeiten können, könnte *Concolic Execution* [20, 45, 11] angewendet werden, um die Eingaben für diese Erweiterungen zu konkretisieren. Dies könnte soweit geführt werden, dass das gesamte Objektsystem bei Bedarf konkretisiert wird, damit eine Transformation mit der *Henshin*-Erweiterung durchgeführt werden kann.

Zudem könnte die symbolische Ausführung dahingehend erweitert werden, dass einige Nachrichtenparameter nicht symbolisch interpretiert werden dürfen. Dies wäre eine ähnliche Interpretation wie bei der *Concolic Execution*. Aus diese Weise könnten Nachrichten gekennzeichnet werden, die von Erweiterungen verarbeitet werden können. Bei diesem Verfahren müsste jedoch geklärt werden, wie stark die symbolische Ausführung durch die Konkretisierung der symbolischen Werte beeinträchtigt wird.

Die Unterstützung von Listenoperationen während der symbolischen Ausführung könnte noch erweitert werden. Dabei könnte beispielsweise die Interpretation von symbolischen Listen eingeführt werden. Diese werden bei der Erkundung als unbekannt betrachtet. Beim Zugriff auf die Liste wird die Ausführung in verschiedene Pfade geteilt. Diese folgen unterschiedlichen Situationen beim Listenzugriff, wie beispielsweise *OutOfBoundsExceptions*.

Khurshid et al. [30] haben dazu bereits ein Konzept beschrieben. Da diese Listen Attribute der Objekte des Objektsystems sind, könnte dabei im gleichem Zuge die symbolische Interpretation des Objektsystems untersucht werden. Interessant wäre hier wie dynamische Rollen mit einem symbolischen Objektsystem gebunden werden können.

Von Kuznetsov [33] wird ein Verfahren vorgeschlagen, bei dem zwei symbolische Zustände, die sich nur in den symbolischen Wertebelegungen unterscheiden, zu einem Zustand zusammengefasst werden. Um eine gute Balance zwischen Rechenzeit und Speicherplatzverbrauch zu sichern, werden verschiedene Metriken berücksichtigt bevor zwei Zustände zusammengefasst werden. Fallen diese Metriken unter einen Schwellenwert, werden die Zustände nicht zusammengefasst. Sind sie über dem Schwellenwert, werden beide Zustände in einem Zustand vereint. Dabei werden die symbolischen Eigenschaften intern durch *if-then-else*-Konstrukte den verschiedenen Zustände zugewiesen.

Aktuell gibt es in ScenarioTools keinen Controller-Synthesealgorithmus, der auf symbolischen Zustandsgraphen arbeiten kann. Die Schwierigkeit eines solchen Verfahrens liegt zum einen darin, dass der symbolische Zustandsgraph eine Überapproximation ist. Zum anderen sind die ausgehenden Transitionen von den Parametern der eingehenden Transitionen abhängig.

Außerdem könnte das bereits in Abschnitt 6.4 besprochene Problem der Integerüberläufe durch die symbolische Ausführung automatisch analysiert werden. Darüber hinaus könnten verschiedene Integer-Größen (8, 16, 32 und 64 Bit) betrachtet werden.

Literaturverzeichnis

[1] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1):53–67, 2009.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proceedings of the 13th International Conference on Model Checking Software*, SPIN'06, pages 163–181, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016. letzer Zugriff: September 2016.

[6] C. Barrett, D. Kroening, and T. Melham. Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories. Technical Report 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, June 2014. Knowledge Transfer Report.

[7] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Frontiers in Artificial Intelligence and Applications*, 185(1):825–885, 2009.

[8] N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning: 5th International Joint Conference,*

IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, pages 316–330. Springer Berlin Heidelberg, 2010.

[9] C. Brenner, J. Greenyer, and V. Panzica La Manna. The scenariotools play-out of modal sequence diagram specifications with environment assumptions. In *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, volume 58. EASST, 2013.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[12] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service. *ACM SIGOPS Operating Systems Review*, 43(4):5, 2010.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. *Computer Aided Verification, 12th International Conference, {CAV} 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[14] L. a. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340. Springer Berlin Heidelberg, 2008.

[16] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[17] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 129–140, New York, NY, USA, 2009. ACM.

[18] H. Felbinger and C. Schwarzl. Suitability analysis of CSP- and SMT-solvers for test case generation. *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis - CSTVA 2014*, pages 40–49, 2014.

[19] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, pages 519–531, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[21] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications (extended abstract of esec/fse 2013 paper). In U. Aßmann, B. Demuth, T. Spitta, G. Püschel, and R. Kaiser, editors, *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, volume 239 of *LNI*, pages 91–92. GI, 2015.

[22] J. Greenyer, D. Gritzner, N. Glade, T. Gutjahr, and F. König. Scenario-based specification of car-to-x systems. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016)*, volume 1559, pages 118–123. CEUR Workshop Proceedings, 2016.

[23] J. Greenyer, D. Gritzner, T. Gutjahr, T. Duente, S. Dulle, F.-D. Deppe, N. Glade, M. Hilbich, F. Koenig, J. Luennemann, N. Prenner, K. Raetz, T. Schnelle, M. Singer, N. Tempelmeier, and R. Voges. Scenarios@run.time – distributed execution of specifications on iot-connected robots. In *Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015*, CEUR Workshop Proceedings, 2015.

[24] J. Greenyer, D. Gritzner, G. Katz, A. Marron, N. Glade, T. Gutjahr, and F. König. Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. *Procedia Technology (Proceedings of the 3nd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering, SysInt 2016)*, (to appear), 2016. 3rd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering (SysInt 2016).

[25] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On composing and proving the correctness of reactive behavior. *2013 Proceedings of the International Conference on Embedded Software, EMSOFT 2013*, 2013.

[26] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, SCESM '06, pages 13–20, New York, NY, USA, 2006. ACM.

[27] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[28] D. Jovanović and L. de Moura. Solving non-linear arithmetic. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 339–354, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] G. Katz. On module-based abstraction and repair of behavioral programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8312 LNCS:518–535, 2013.

[30] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. *Tools and Algorithms for the Construction and Analysis of Systems*, 2619:553–568, 2003.

[31] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[32] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2010.

[33] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[34] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[35] V. P. L. Manna, I. Segall, and J. Greenyer. Synthesizing tests for combinatorial coverage of modal scenario specifications. In

Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on, pages 126–135, Sept 2015.

[36] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. *SIGPLAN Not.*, 37(11):83–100, 2002.

[37] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. *Frontiers in Artificial Intelligence and Applications*, 185(1):131–153, 2009.

[38] Y. V. Matijasevich. *Hilbert's Tenth Problem*. The MIT Press, 1994.

[39] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[40] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. *Proceedings of the 2008 international symposium on Software testing and analysis ISSTA 08*, pages 15–26, 2008.

[41] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[42] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[43] N. Rapin. Symbolic execution based model checking of open systems with unbounded variables. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5668 LNCS:137–152, 2009.

[44] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag.

[45] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[46] A. Solar-Lezama. MIT 6.858 Fall 2014: Computer Systems Security, Lecture: Symbolic Execution. `http://css.csail.mit.edu/6.858/2014/`, 2016. letzer Zugriff: September 2016.

[47] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 37–48, New York, NY, USA, 2006. ACM.

[49] T. Wang, A. Roychoudhury, R. H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004. Proceedings*, pages 178–192, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[50] K. Zurowska and J. Dingel. Symbolic execution of UML-RT State Machines. *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, page 1292, 2012.

Anhang A

Anhang

A.1 Ofensteuerung

Listing A.1 zeigt die vollständige Spezifikation der Ofensteuerung in Variante *Wmax*, wie sie in der Evaluation 6.2 verwendet wurde. Die Konfiguration für eine symbolische Ausführung ist in Listing A.2 dargestellt. Das Objektsystem mit dem die Spezifikation ausgeführt wurde ist in Abbildung A.1 zu sehen.

```
1  import "../model/oven.ecore"
2
3  system specification OvenSpecification {
4
5    domain oven
6
7    define Controller as controllable
8    define TemperatureSensor as uncontrollable
9    define HumiditySensor as uncontrollable
10   define Heater as uncontrollable
11   define Panel as uncontrollable
12
13   parameter ranges {
14     Controller.reportTemperature(tmp = [ 0 .. 220]),
15     Controller.reportNominalTemperature(nominalTmp = [ 0 .. 220 ]),
16     Controller.reportHumidity(humidity = [ 0 .. 100 ])
17   }
18   collaboration OvenCollaboration {
19
20     static role Controller ctr
21     static role TemperatureSensor ts
22     static role HumiditySensor hs
23     static role Heater heater
24     static role Panel panel
25
26     specification scenario HumindityRegulation {
27       var EInt humidity
28       message hs -> ctr.reportHumidity(bind to humidity)
29       interrupt if [ ctr.ovenStatus == false ]
30       interrupt if [ ctr.temperature < 50 ]
31       alternative if [humidity > 50]{
32         message strict requested ctr -> ctr.lessHumidity()
33       } or if [ humidity <= 50]{
34         message strict requested ctr -> heater.moreHumidity()
35       }
36     }
37
38     specification scenario OvenRegulation {
39       var EInt tmp
40       message ts -> ctr.reportTemperature(bind to tmp)
```

```
41        interrupt if [ ctr.ovenStatus == false ]
42        message strict requested ctr -> ctr.setTemperature(tmp)
43        alternative if [tmp > ctr.nominalTemperature]{
44          message strict requested ctr -> heater.turnOff()
45        } or if [ tmp <= ctr.nominalTemperature]{
46          message strict requested ctr -> heater.turnOn()
47        }
48      }
49
50      specification scenario NominalTemperature {
51        var EInt nominalTmp
52        message panel -> ctr.reportNominalTemperature(bind to nominalTmp)
53        message strict requested ctr -> ctr.setNominalTemperature(nominalTmp)
54      }
55
56      specification scenario PreheatLightOn {
57        var EInt tmp
58        message ts -> ctr.reportTemperature(bind to tmp)
59        interrupt if [ tmp > ctr.nominalTemperature
60        | ctr.ovenStatus == false ]
61        message strict requested ctr -> ctr.preheatingLight(Status:ON)
62      }
63
64      specification scenario PreheatLightOff {
65        var EInt tmp
66        message ts -> ctr.reportTemperature(bind to tmp)
67        interrupt if [ tmp <= ctr.nominalTemperature
68                  | ctr.ovenStatus == false ]
69        // error
70  //    interrupt if [ tmp < ctr.nominalTemperature
71  //              | ctr.ovenStatus == false ]
72        message strict requested ctr -> ctr.preheatingLight(Status:OFF)
73      }
74
75      specification scenario OvenOn {
76        message panel -> ctr.oven(Status:ON)
77        interrupt if [ ctr.ovenStatus == true ]
78        message strict requested ctr -> ctr.setOvenStatus(true)
79      }
80
81      specification scenario OvenOff {
82        message panel -> ctr.oven(Status:OFF)
83        interrupt if [ ctr.ovenStatus == false ]
84        message strict requested ctr -> heater.turnOff()
85        message strict requested ctr -> ctr.setOvenStatus(false)
86      }
87
88      assumption scenario DeltaTemperature {
89        var EInt tmp
90        message ts -> ctr.reportTemperature(bind to tmp)
91        alternative {
92          message strict requested ctr -> heater.turnOn()
93          message strict requested ts -> ctr.reportTemperature(tmp + 1)
94        }or{
95          message strict requested ctr -> heater.turnOff()
96          message strict requested ts -> ctr.reportTemperature(tmp - 1)
97        }
98      }
99  }// end colloboration
100 }// end specification
```

Listing A.1: Spezifikation der Ofensteuerung

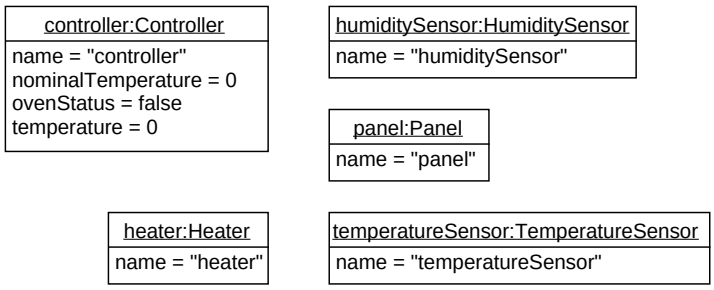

Abbildung A.1: Vollständiges Objektdiagramm der Ofensteuerung

```
1  import "oven.sml"
2  configure specification OvenSpecification
3  use instancemodel "oven.xmi"
4
5  rolebindings for collaboration OvenCollaboration {
6    object oven.heater plays role heater
7    object oven.temperatureSensor plays role ts
8    object oven.humiditySensor plays role hs
9    object oven.controller plays role ctr
10   object oven.panel plays role panel
11 }
12
13 symbolic parameters {
14   Controller.reportTemperature(tmp:symbolic),
15   Controller.reportNominalTemperature(nominalTmp:symbolic),
16   Controller.reportHumidity(humidity:symbolic)
17 }
```

Listing A.2: Konfiguration der Ofensteuerung

A.2 Tunnelsteuerung

Das Listing A.3 zeigt die vollständige Spezifikation der Tunnelsteuerung in Variante *Wmax*, wie sie in der Evaluation 6.1 verwendet wurde. Die Konfigurationsdatei in Listing A.4 zeigt die Konfiguration für eine symbolische Ausführung.

```
1  import "../model/tunnelcontrol.ecore"
2
3  system specification TunnelSystemSpecification {
4
5    domain tunnelcontrol
6
7    define CarApproachSensor as uncontrollable
8    define EntryExitSensor as uncontrollable
9    define TrafficLight as uncontrollable
10
11   define TunnelSystem as controllable
12   define Gate as controllable
13
14 // channels beschreiben für einzelne Objekte mit welchen
15 // anderen Objekten sie kommunizieren können
16 channels{
```

```
17    Gate.entry over EntryExitSensor.gate
18    Gate.exit over EntryExitSensor.gate
19    Gate.carApproaching over CarApproachSensor.gate
20  }
21
22  parameter ranges {
23    TunnelSystem.setMaxNumberOfCarsInTunnel(maxNumberOfCarsInTunnel = [0..100]),
24    Gate.setCarsInTunnel(carsInTunnel = [0..101])
25  }
26
27  collaboration IncrementGateCounterWhenCarEntersTunnel {
28
29    dynamic role EntryExitSensor entryExitSensor
30    dynamic role Gate gate
31    dynamic role TrafficLight trafficLight
32    dynamic role TunnelSystem tunnelSystem
33
34    specification scenario IncrementGateCounterWhenCarEntersTunnel{
35      message entryExitSensor -> gate.entry()
36      var EInt newValue = gate.carsInTunnel+1
37      message strict requested gate -> gate.setCarsInTunnel(newValue)
38    }
39
40    assumption scenario NoCarEntersWhenTrafficLightShowsStop
41      with dynamic bindings [bind trafficLight to gate.trafficLight]
42    {
43      message entryExitSensor -> gate.entry()
44      violation if [trafficLight.showStop]
45    }
46
47    assumption scenario EntryExitSensorOnlySendsEntryToLocalGate{
48      message entryExitSensor -> gate.entry()
49      violation if [gate.entryExitSensor != entryExitSensor]
50    }
51
52    assumption scenario MaxNumberOfCarsInTunnel
53      with dynamic bindings [bind tunnelSystem to gate.system]
54    {
55      message entryExitSensor -> gate.entry()
56      violation if [gate.carsInTunnel > tunnelSystem.maxNumberOfCarsInTunnel]
57    }
58  }
59
60  collaboration DecrementGateCounterWhenCarExitsTunnel {
61
62    dynamic role EntryExitSensor entryExitSensor
63    dynamic role Gate gate
64    dynamic role Gate oppositeGate
65
66    specification scenario NotifyOppositeGateWhenCarExitsTunnel
67      with dynamic bindings [bind oppositeGate to gate.oppositeGate]
68    {
69      message entryExitSensor -> gate.exit()
70      message strict requested gate -> oppositeGate.carLeftTunnel()
71    }
72
73    specification scenario DecrementGateCounterWhenCarLeftTunnel{
74      message gate -> oppositeGate.carLeftTunnel()
75      var EInt newValue = oppositeGate.carsInTunnel+-1
76      message strict requested oppositeGate
                                      -> oppositeGate.setCarsInTunnel(newValue)
77      violation if [oppositeGate.carsInTunnel < 0]
78    }
79
80    assumption scenario NoCarLeavesWhenTunnelEmpty with dynamic bindings [
81    bind oppositeGate to gate.oppositeGate
82    ]{
83      message entryExitSensor -> gate.exit()
84      violation if [oppositeGate.carsInTunnel == 0]
85    }
86
87    assumption scenario EntryExitSensorOnlySendsExitToLocalGate{
88      message entryExitSensor -> gate.exit()
89      violation if [gate.entryExitSensor != entryExitSensor]
```

```
90    }
91
92  }
93
94  collaboration CarApproachesGateWithStopLight {
95
96    dynamic role CarApproachSensor carApproachSensor
97    dynamic role Gate gate
98    dynamic role Gate oppositeGate
99    dynamic role TrafficLight trafficLight
100   dynamic role TrafficLight oppositeTrafficLight
101
102   singular specification scenario CarApproachesGateWithStopLight
103     with dynamic bindings [
104       bind oppositeGate to gate.oppositeGate
105       bind oppositeTrafficLight to oppositeGate.trafficLight
106       bind trafficLight to gate.trafficLight
107   ]{
108     message carApproachSensor -> gate.carApproaching()
109     // it's green, nothing to request
110     interrupt if [!trafficLight.showStop]
111     // check if someone else started a request already
112     interrupt if [oppositeTrafficLight.showStop & trafficLight.showStop]
113     // we must close the opposite gate and open this gate when all
114     // cars have left the tunnel.
115     message strict requested gate -> oppositeGate.close()
116     message strict requested oppositeGate
                                    -> oppositeTrafficLight.setShowStop(true)
117     strict wait until [oppositeGate.carsInTunnel == 0]
118     message strict requested oppositeGate -> gate.tunnelIsEmpty()
119     message strict requested gate -> trafficLight.setShowStop(false)
120   }constraints [
121     ignore message carApproachSensor -> gate.carApproaching()
122   ]
123 } // end collaboration
124 } // end specification
```

Listing A.3: Spezifikation der Tunnelsteuerung

```
1   import "TunnelSystem.sml"
2   configure specification TunnelSystemSpecification
3   use instancemodel "TunnelSystem.xmi"
4
5   symbolic parameters {
6     Gate.setCarsInTunnel(carsInTunnel:symbolic)
7   }
8
9   symbolic attributes {
10    TunnelSystem.setMaxNumberOfCarsInTunnel(maxNumberOfCarsInTunnel:symbolic),
11    Gate.setCarsInTunnel(carsInTunnel:symbolic)
12  }
```

Listing A.4: Konfiguration der Tunnelsteuerung

A.3 Kaskade Variante A

Listing A.5 zeigt die Spezifikation der Kaskade in *Variante A*, wie sie in der Evaluation 6.3 verwendet wurde.

```
parameter ranges {
  A.startCascade(value = [ 25 ]) // 0..50, 0..100, 0..100000
}
...
// Startkaskade
specification scenario Cascade1 {
  var EInt value
  message e->a.startCascade(bind to value)
  message strict requested a->b.msg1(value)
  message strict requested a->b.msg1(value)
}

// Variante A
specification scenario Cascade2 {
  var EInt value
  message a->b.msg1(bind to value)
  message strict requested a->b.msg2(value + 1)
  message strict requested a->b.msg2(value + 1)
}

specification scenario Cascade3{
  var EInt value
  message a->b.msg2(bind to value)
  message strict requested a->b.msg3(value + 1)
  message strict requested a->b.msg3(value + 1)
}

specification scenario Cascade4 {
  var EInt value
  message a->b.msg3(bind to value)
  message strict requested a->b.msg4(value + 1)
  message strict requested a->b.msg4(value + 1)
}

specification scenario Cascade5 {
  var EInt value
  message a->b.msg4(bind to value)
  message strict requested a->b.msg5(value + 1)
  message strict requested a->b.msg5(value + 1)
}
```

Listing A.5: Vollständige Kaskade der Variante A

A.4 Inhalt der DVD

Im Folgenden ist der Inhalt der beigelegten DVD aufgelistet:

- Ausarbeitung im PDF Format
- VirtualBox Image einer Ubuntu 15 Installation. Darauf befindet sich eine Installation von ScenarioTools mit den in dieser Arbeit erstellten Erweiterungen.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 30.09.2016

Timo Gutjahr